Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Mauri Flinckman

Flexible Data Synchronization for Supporting Mobile Offloading Applications

Master's Thesis
Espoo, May 21, 2015

Supervisor: Assoc. Prof. Keijo Heljanko
Advisor: Assoc. Prof. Keijo Heljanko

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Mauri Flinckman

Title:
Flexible Data Synchronization for Supporting Mobile Offloading Applications

Date: May 21, 2015 **Pages:** 79

Major: Software Engineering and Business **Code:** T-110

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: Assoc. Prof. Keijo Heljanko

This Thesis surveys mobile offloading approaches and data synchronization methods. A series of tests to offload face recognizer application using Apache Spark framework with different levels of data synchronization is conducted. We evaluate the application's performance in terms of time and energy, as well as average memory and CPU utilization. In addition to offloading the entire application, we perform tests that run the application in parallel locally on the mobile device and remotely at the server. Finally, we evaluate the synchronization methods performance for a large directory synchronization.

The chosen synchronization methods in this Thesis were Git and Rsync. Git outperformed Rsync in terms of computing the list of changed files since last synchronization round, while Rsync was able to synchronize the test data sets used in the face recognizer application and the Linux distributions using less time and energy than Git. Git supports a larger feature set compared to Rsync and is more applicable for tasks that require two-way synchronization in a mobile device environment.

Offloading the task of face recognition on Asus Nexus 7 2013 tablet with Ubuntu Touch operating system had over ten times benefit in terms of energy expenditure compared to local execution. For the partial offloading tests, energy expenditure were higher compared to performing data set synchronization and offloading the application to the server. On the other hand, in terms of running time with using one local worker and one remote worker, 90% partial offloading had the shortest execution time.

Test runs with suspension mode revealed that the tablet's energy efficiency was improved when low workload tasks were run in suspension mode, while the opposite was observed for high workload tasks. As data synchronization of large data sets were measured to be highly intensive, it is advisable to perform high workload tasks in active CPU power state while using maximum number of the mobile device's available CPUs.

Keywords: mobile offloading, data synchronization, Apache Spark, Git, Rsync

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Mauri Flinckman		
Työn nimi:	Joustava datan synkronointi mobiilisovellusten laskennan siirtoa varten		
Päiväys:	21. toukokuuta 2015	**Sivumäärä:**	79
Pääaine:	Ohjelmistotuotanto ja -liiketoiminta	**Koodi:**	T-110
Valvoja:	Professori Keijo Heljanko		
Ohjaaja:	Professori Keijo Heljanko		

Tämä työ tekee katsauksen laskennan siirtomenetelmiin sekä datan siirtomenetelmiin mobiiliympäristössä. Kasvojentunnistus sovelluksen laskennan siirtoa varten ajetaan sarja testejä Apache Spark datan prosessointi -kehysohjelmistolla. Arvioimme sovelluksen suorituskyvyn suhteessa käytettyyn aikaan ja energiaan, sekä keskimääräisen muistin ja prosessorien käyttöasteen. Koko sovelluksen laskennan siirron lisäksi ajamme testejä, jotka suorittavat laskentaa rinnakkain sekä paikallisesti mobiililaitteessa että palvelimella. Lopuksi arvioimme synkronointimenetelmien suorituskyvyn laajan tiedostohakemiston synkronoimiseen.

Datan synkronointimenetelmiksi valittiin Git ja Rsync. Git muodosti muuttuneiden tiedostojen listan viimeksi tehdystä synkronoinnista nopeammin kuin Rsync, kun Rsync taas käytti testidatan synkronointiin sekä kasvojentunnistus sovelluksessa että Linux ytimen testeissä vähemmän aikaa ja energiaa. Git tukee useampaa ominaisuutta kuin Rsync ja on näin sopivampi tehtävissä, jotka edellyttävät kaksisuuntaista synkronointia mobiiliympäristössä.

Kasvojentunnistus sovelluksen laskennan siirto Asus Nexus 7 2013 tabletilla Ubuntu Touch käyttöjärjestelmällä käytti alle kymmenysosan energiaa verrattuna paikalliseen laskentaan. Osittaislaskennan testien kohdalla energian kulutus oli suurempi kuin datan synkronointiin ja sovelluksen laskennan siirtoon palvelimelle käyttämä kokonaisenergia. Toisaalta, suurin suorituskyky ajan suhteen saavutettiin 90%:n osittaisella laskennan siirrolla.

Virransäästötilassa ajetut testit osoittivat, että tabletin energiatehokkuus parani, kun matala intensiiviset laskennan siirtotestit suoritettiin tässä tilassa. Toisaalta ajettaessa korkea intensiivisiä laskennan siirtotestejä, energiatehokkuus kasvoi käytettäessä tabletin aktiivitilaa. Datan synkronointi kokeissa ilmeni myös vaiheita, jotka käyttivät suuren osan laitteen laskentaresursseista. Näin on suositeltavaa suorittaa korkea intensiiviset tehtävät laitteen ollessa aktiivitilassa käyttäen saatavilla olevien suorittimien resursseja.

Asiasanat:	laskennan siirtomenetelmät mobiiliympäristössä, datan synkronointi, Apache Spark, Git, Rsync
Kieli:	Englanti

Acknowledgements

I would like to express my gratitude for my supervisor, Keijo Heljanko, for giving me direction during the time of conducting experiments and throughout writing this work. In addition, I would like to say thank you to Markus Losoi, who provided me modified Apache Spark framework and face recognition application and instructed me to use the framework. Finally, I want to thank my family who supported me in various ways.

Espoo, May 21, 2015

Mauri Flinckman

Abbreviations and Acronyms

3G	3rd Generation of Mobile Telecommunications Technology
Btrfs	B-tree File System
CoW	Copy-on-Write
DEM	Device Elasticity Manager
DSM	Distributed Shared Memory
EXI	Efficient XML Interchange
GPU	Graphics Processing Unit
HTTP	Hypertext Transport Protocol
MD5	Message Digest 5
MRCA	Most Recent Common Ancestor
OS	Operating System
PSM	Power Saving Mode
QR code	Quick Response code
RDD	Resilient Distributed Dataset
REST	Representational State Transfer
RMI	Remote Method Invocation
RSA	Rivest-Shamir-Adleman Crypto System
RTT	Round Trip Time
TCP	Transmission Control Protocol
UDP	Used Data Protocol
VM	Virtual Machine
WLAN	Wireless Local Area Network
XML	Extended Markup Language
YARN	Yet Another Resource Negotiator

Contents

Chapter 1

Introduction

Both storage capacity and computational power has developed in mobile devices to a point that enables users to smoothly produce and view multimedia content and personal documents using their personal mobile phones, tablets, and laptops. Furthermore, certain devices are not designed for high computation demanding tasks, and instead must rely on offloading the computationally intensive tasks. Examples are appliances equipped with sensor devices collecting measurements from the environment or with camera and head-up display, such as Google Glass. While the needs of computationally demanding application have increased, available battery capacity has not increased relative to advancements in computing capacity. Most of the changes in battery technology have impacted on other aspects than battery capacity, such as recharging and memory effects [33].

Keeping users data, such as music, videos, books, pictures, documents and applications synchronized in a mobile environment with users fixed and auxiliary mobile devices, such as notebooks, may become laborious with manual synchronizing of data sources. Synchronization involves the detection of changed files, the computing of changes for each modified file between the user's mobile device and target device, the compression of these deltas, the transfer of the deltas, and finally the reconstruction of the files. Furthermore, many synchronization tools also perform data integrity checking and encryption. While synchronization process requires energy in terms of detecting or logging changes, the overall energy usage favors the use of synchronization tool over the whole directory files or individual file for each synchronization round.

Data synchronization may require the support of making two types of synchronization. In the first type called one-way synchronization, the mobile device stores the master replica of the data source and other devices synchronize with the mobile device data source. The typical policy of one-way

synchronization is to keep the changes of the newer version of the source or invoke a tool to manually decide which versions changes are kept in the case the mobile device has older files compared to the target device. In the second type of synchronization both the source and the target may have produced changes to the same files and their changes are merged. Merging takes as inputs both versions of the local file and remote file and a base version of the file [50]. The base file is the earliest version of the file after which the two file versions have diverged and it is called the most recent common ancestor (MRCA) [48]. This approach requires to maintain a copy of each file synchronized on the previous time, which the user wants to be able to merge. Finally, in case the files have changes at the same location, the conflict situation can be resolved manually by choosing the modifications from one of the files that are included into the new merged version.

The decision to perform data synchronization is influenced with current environmental conditions which have an effect to the speed and energy efficiency of the process. Latency is the total time to send and receive a reply from the destination. With higher end-to-end communication time, the sender must keep the radio receiver longer open or use a power saving mode (PSM) to switch the radio off and wake it back in predetermined frequency. In [33], the authors tested to send a packet load of 500KB in two experiments using round trip times (RTT) of 75 ms and 25 ms while enabling the PSM. The authors found that with the 75 ms RTT, the PSM decreased energy usage due to the being able to send all packets before sleep while waiting for the ACKs. On the other, in experiment using 25 ms RTT, the power draw increased due to PSM increasing the transfer time.

Another factor affecting the energy efficiency of data transfer is bit rate. Authors in [53] evaluate Nokia N95 bit rate to energy efficiency for both wireless local area network (WLAN) and 3rd generation mobile telecommunication technology (3G). They mention that most significant factor affecting the energy consumption of wireless modem is the activity time of the interface. They continue that in 3G latencies between the activation and deactivation of the wireless modem are longer with 3G than WLAN. The authors report that energy efficiency of data transfer is higher with higher bit rate. In addition, 3G tends to be more sensitive to bit rate. In their experiment, WLAN had higher energy efficiency for each transferred bit compared to 3G.

Kosta et al. [43] use both a local WLAN with low latency time and a public WLAN hot spot with around 200 ms RTT. They conduct experiments for mobile offloading using both WLAN connections and 3G to connect to a cloud server. With higher data transfer amounts in face detection and virus scanning 3G's energy usage was similar or lower than energy used with the public WLAN.

The module that makes the offloading decision takes as inputs the above mentioned factors. Other direct factors affecting to data transfer cost are service cost and required level of security [30]. Furthermore, the offloading decision module estimates local computation cost and remote computing cost of the offloaded part of the application. In elastic application model as proposed in [67], Maui in [33] and ThinkAir in [43] as well as in Cuckoo in [57], the offloading framework provide an application programming interface (API) for marking offloadable methods as remote and adding necessary wrappers for these methods in compiling phase. During run time the framework calls the profiler or offloading decision module to estimate the offloadable methods execution time usage and energy consumption. As run time conditions vary, the offloading decision may be computed several times during different executions of the application.

In the Elastic application model the application is composed of UI components and weblets which are autonomous software entities whose properties, such as computation requirements, are described in the application's manifest. These properties can be used to direct the offloading decision module. When deciding to offload a weblet, the elastic application model's run time releases any resources hold by the local weblet and routes any communication for the weblet to the host cloud system. Communication between the weblets and UI is implemented with hypertext transport protocol (HTTP). As each weblet is designed as an autonomous unit, each weblet may be decided to be deployed to a cloud service.

Another design approach is taken in Maui which handles application's state synchronization by transferring the state of the entire application to the server. It uses .NET reflection to serialize parameters similar to ThinkAir's Java reflection. To optimize the size of the state of the application, incremental changes of the application state are computed relative to the last time the method was called. The entire application needs to be available at the server. The code may be transferred from the mobile device to the server or the server can receive a signature from the phone and download the application from a cloud service [33].

In this Thesis the focus is on studying offloading face recognition using different types of synchronization methods and their effect on the mobile device's total energy consumption and performance. Furthermore, partial offloading is experimented with computing in parallel with new pictures on the phone and offloading computation for previously transferred pictures. We compare the two approaches and perform the synchronization using Rsync and Git. The mobile device used in this work is the Asus Nexus 7 2013. The face recognition application used for conducting the experiments and the modified Apache Spark used as the offloading framework are based on

the work in [51].

The structure of this Thesis is following. In Section 2 we make a survey of mobile offloading approaches and give example applications using data synchronization and mobile computation offloading. In Section 3 we introduce several data synchronization methods with a focus on one-way synchronization tools. Section 4 contains the results and evaluation of face recognition application on a modified Apache Spark and Linux distribution synchronization in Asus Nexus 7 2013 tablet using both Git and Rsync. Section 5 includes discussion and conclusions focusing on data synchronization energy efficiency on the mobile device.

Chapter 2

Background

This chapter presents a mobile offloading cost model and analyzes energy efficient computing. We summarize ways for implementing mobile offloading and go through selected mobile offloading applications. Finally, we evaluate the effect of storing data in structured format on data synchronization process, energy efficient computing for variable task load levels and give an overview of file directory tree update detection method.

2.1 Mobile Offloading

This chapter discusses mobile offloading in the context of migrating computationally demanding applications or their components to a remote server. Another closely related concept is mobile data offloading. In this approach the data is routed through alternative networks to the target mobile devices, such as when offloading mobile traffic from the mobile network service providers cellular system to local wireless access points [34] or propagating content to a set of key users from which other mobile users may choose to download content [38].

The focus on mobile offloading has changed over the decades [44] from studying the feasibility of offloading, shifting to offloading infrastructure and finally taking service viewpoint with the advent of cloud systems. The authors in [44] present as enabling factors wireless networks, virtualization and cloud systems. Until late 1990s, low bandwidth wireless networks where the main problem for mobile systems. The focus on building wireless data networks has facilitated mobility. Virtualization is a technique which allows to run several instances of virtual machines with their own of operating system (OS) in one physical computer. Guest machine is referred as virtual machine and host machine is the computer which runs the guest machines.

The main benefits of virtualization include high utilization of computing resources and the ability to launch new virtual machine instances with respect to the activity and number of clients.

Kumar et al. overview algorithms for making the offloading decision which they divide into static and dynamic [44]. The dynamic algorithms typically take into account several factors, such as bandwidth, available resources in both local and offloading computation nodes, network connection type and the amount of data exchanged between the server and mobile node. While dynamic decision making has a higher cost between the two methods, it may adopt more flexible to changing environment conditions, such as fluctuation in available network bandwidth. Typical clients utilizing offloading according to Kumar et al. are smartphones (60%), laptops (20%), embedded boards (13.3%) and robots (6.7%).

The decision to offload application or partitions of it requires to inspect mainly two variables, communication and computation/task complexity. The article [45] divides the offloading decision to three confidence category based on the relationship between cost factors. The first case, to never offload, occurs when the need for computing is relatively low and/or the amount of communication is costly. The second case, always offload, is seen feasible when the amount of computing needed is high compared to the amount of communication. Between these two territories lies the area where the decision depends on the quality of the datapath (bandwidth and latency) and amount of required two way communication trips.

Chen et al. in [30] estimate the amount of computing intensity required per transferred data unit (32 bit) in order for the offloading to be feasible. They come to different recommendations for outsourcing criterion depending on the target organization's characteristics. For home users they recommend tasks that require at least 1000 cycles per 32 bit in order to generate savings. For small, medium and large organizations the corresponding numbers varies due to different requirements. The authors compute the threshold requirement for medium organizations to move their workload on cloud to be 96,000 cycles per 32 bit due to network costs being higher for guaranteed bandwidth. Finally, they assess the criterion to outsource in terms of required level of security and estimate costs of common data securing techniques, such as RSA and MD5. In addition they consider the case where the local data storage is deployed to the cloud as encrypted file system and evaluate costs for existing secure data query mechanisms, which employ techniques such as hash trees and aggregated signatures. They found a borderline for outsourcing simple range queries, which involve large amount of data tuples and execute selective queries that return only a fraction of the data set.

2.1.1 Cloudlets

Bahtovski and Gusen discuss the concept of cloudlets in [23]. Long distance to the deployed clouds increases latency and network service costs and may reduce user experience of real time applications. As an example, Amazon's cloud infrastructure is located in six cities [23]. To mitigate the effect of round trip communication costs, the cloud can be deployed in smaller scale to near vicinity of mobile users by decreasing the number of nodes. Thus, a cloudlet is a small cloud (less than 10 concurrent users) that its clients may use by either instantiating custom virtual machines to the cloudlet or leveraging earlier deployed virtual machines. Internally cloudlets are data centers with multiple computation nodes connected through high bandwidth networking equipment. Mobile users access the cloudlet's services for instance by connecting to web services built with representational state transfer (REST) architecture and running the application software as a thin client. A user may disconnect from the cloudlet and continue their task by connecting to a distant cloud or use the mobile device's own resources.

In the architecture proposed in [23] cloudlets may be connected to distant clouds. In this model, the distant cloud may be implemented as a private cloud or as a public cloud, such as Amazon's data center or Google's data center. A hybrid cloud may also be seen as a viable alternative. The communication to the distant clouds requires several network hops while a single network hop is required for the mobile clients to connect to the cloudlet. Mobile devices are connected to the cloudlets by short-range radio or WiFi.

The method of forming a cloudlet by exploiting nearby mobile devices to act as a virtual cloud computing provider is studied in [39]. The authors note that co-location increases the opportunity of sharing common goals or activities which acts as an incentive to share resources among users in the same area. In order to implement ad hoc cloudlet Huerta et al. propose an architectural model to support a virtual cloud computing platform. The first element is a resource manager which profiles the application and assess the required number of devices needed to complete the application's goal. Second component is offloading manager which is responsible of managing jobs received from other nodes and job submission to other nodes. The architecture also requires a method for maintaining information of the nearby devices. This is managed by peer-to-peer (P2P) component which detects new devices entering to the area of the mobile device or leaving its area.

2.1.2 Approaches

Kemppainen gives a taxonomy of mobile offloading with three categories [42]. In the first category, feature offloading, semantically coherent parts of the application are offloaded. The application is designed in service oriented architecture like in Cuckoo and elastic application model, and thus, the client transfers only the service's remote implementation over to the cloud server. The second category, called method offloading, is technically similar to feature offloading, but requires that the server has a full copy of the application. In method offloading both application state and method parameters are synced to the server. Microsoft's Maui framework is given as an example. Lastly, offloading can be implemented by transferring the application image to the offloadable computation site. This is called image offloading. An example implementation is presented in [58] where the authors perform offloading by sending an application overlay to a remote cloudlet, which synthesizes the overlay image with the preinstalled base virtual machine image that contains the OS.

2.1.3 Frameworks

2.1.3.1 Elastic Application Model

Zhang et al. propose an application model and offloading framework in [67], in which the application is composed of application root and independently deployed components called weblets. Weblets expose their services through REST interface and they can be deployed either at the users mobile phone or at the cloud service provider's infrastructure. The application root includes the user interface (UI) and is connected to the weblets through HTTP(S). With each application comes a manifest which contains information about the application's requirements and the individual weblets. Such information may contain requirements for compute power, memory need, network and storage usage and maximum instances of the weblet that can be launched.

The architecture consists of two major components. On the device side, the key component is device elasticity manager (DEM), which decides, for instance, where the application components are located and which communication paths are used (3G/4G/WiFi). The device also provides sensing information, such as battery levels and processor types, which are used to decide when and where a new weblet instance should be deployed.

On the cloud's side, the key component is cloud elasticity service (CES), which consists of cloud manager, application manager and sensing information component. The cloud manager allocates and releases the cloud's re-

sources for the clients. In addition, it may track the cloud resource usage by receiving history data from the sensing information component. The application manager is responsible for installing and managing weblets, which are run in weblet containers in a cloud node. Each cloud node inlcudes a node manager that can run one or several weblet containers in an Amazon EC2 instance [67]. The node manager communicates with the cloud manager and application manager.

2.1.3.2 Cuckoo

Cuckoo is an example of mobile feature offloading framework developed for Android [57]. Cuckoo's programming model uses Android's activity/service model. Activities contain the UI elements while services perform computing intensive parts. The developer designs the interface and realizes the implementations for the offloadable parts of the application. Remote implementation may differ from the local service implementation, for example by using a higher fidelity algorithm or using hardware acceleration. Resources must be registered to the Cuckoo framework after which any applications built with Cuckoo can use the resource. Registering may be done either by scanning an quick response (QR) code or by manually copying a resource description file over to the phone. During run time, Cuckoo's resource manager makes an offloading decision for any method that has been annotated as to be offloaded. The build process instruments the source code for placing appropriate instructions in the source code which make calls to the resource manager.

For communication between the server and client, Cuckoo uses Ibis middleware implemented in Java. Ibis distributed deployment system deploys the client's application to the remote server, when the server does not include the service. For message passing, Ibis implements several communication protocols, such as transmission control protocol (TCP), user data protocol (UDP) and Bluetooth. Ibis uses the concept of send and receive ports which may be connected in one-to-one, many-to-one or many-to-many mode. Thus, the server may be used by several clients.

In Cuckoo, client may be configured to either bind early or late to the remote service. Early binding occurs at the time when the client binds to the remote service, thus allowing to make immediately remote service invocations. This option favors speed. On the other hand, late binding is invoked during the call of the remote service. This approach can save energy which is created when the client needs to reconnect to the distant server due to the close of connection channel.

2.1.3.3 CloneCloud

Intel developed CloneCloud, which supports migration of applications threads running within a virtual machine, such as Java, Dalvik virtual machine (VM) in Android and Microsof's .NET, to the device's clone in cloud [31, 32]. The reason for targeting applications running in virtual machine is the ease of modifying application's executable and also migrating to diverging hardware architectures.

Partitioning mechanism in CloneCloud outputs a partition, which is a set of execution points where the application's thread is migrated to the clone. Several partitions are created according to configuration parameters that vary in terms of migrated device's platform, network conditions and optimization constraints for running time and expended energy. For selecting a suitable partition in accordance with current environment conditions, the partitions are stored to database.

Partitioning consists of three phases. In the first phase, the application is analyzed statically to identify proper points of migration from the executable. These locate in the method entry points and exit points. Methods that access the native state of the device, such as global positioning system (GPS) or camera, cannot be migrated, though the authors suggest an alternative approach which places remote method invocation (RMI) calls in place of methods accessing native state. Secondly, the application is run both in the local device and clone environment and its methods execution is profiled in terms of time and for the migrated methods in terms of size of thread state upon migration and when returning from the method invocation. The dynamic profiler select inputs randomly for the executions runs and runs the application several times in both environments. For each execution run a profile tree records in its nodes the running time of the respective method and the edges the cost of migrating and resuming the edge's child node. In the final phase, the profile trees are used to create a cost model, which can be used for selecting a suitable partition with respect to optimization constraints.

In CloneCloud energy consumption is approximated with a model which maps central processing unit (CPU) activity (idle, active), display (on/off) and network state (transmitting/receiving/idle) to evaluate the device's power usage. For local execution the configuration is set to CPU active with the network idle and screen on and for execution on the clone CPU idle with network on and display on.

In an example scenario, the mobile device runs an application's UI thread and migrates a worker thread to the clone. The part of the application's state reserved for the thread is marked and a thread accessing that part

of the thread causes suspension of the the thread. Threads on the clone may use non-virtualized resources, such as graphics processing unit (GPU) for rendering. The migration process is assisted by per process migrator thread, which tasks involve packaging, suspending, resuming and merging of the thread state. In addition to the partition database and migrator assistor running outside the application virtual machine, a per node manager synchronizes the migrated thread's state between the the device and the clone. At the clone a new thread is created for each migrated thread. Once the thread execution is complete at the clone, it's state is synchronized back to the device, where the state of the device's thread is updated by merging it with the state of the migrated thread. The device may then continue the application locally until the execution transfers to a point where one of the application thread is once again migrated.

2.1.3.4 Comet

Comet implements distributed shared memory (DSM) to synchronize the state of the VMs and to continue execution of a thread at the server without requiring that the programmer marks methods offloadable [36]. The synchronization is done at field level granularity by tracking the dirty status of each field. The choice of using DSM allows to migrate threads at any point of its execution path and support multi-threading. In order to offload concurrent programs, Comet tracks the lock owner of each object, which indicates the endpoint that last held the lock. When a thread wishes to acquire a lock, which its endpoint does not currently own, the thread can be migrated or it can make a request for the ownership of the lock.

The authors of Comet in [36] present their evaluations results for 8 interactive applications downloaded from Google Play store and two benchmark application, both computation benchmarks which one also utilizes multi-threading for solving integer factors. As a test phone, the authors used Samsung smartphone running Android 2.3.4 with extended Dalvik VM. The phone was connected to the authors cloud server conducting the experiments for both WiFi and 3G. The average performance gain using WiFi for the interactive applications were 2.8X, while using 3G only two applications gained improvements in terms of performance with 1.28X average mean due to high latency. Respective energy consumptions gain were 1.5X for WiFi and 0.84X for 3G. Comet presented the gain of supporting multi-threading by improving the performance of the computational benchmarks solving the problem concurrently with 8 threads by 202X and 168X for WiFi and 3G respectively. In terms of energy consumption, this yielded 517X and 185X gains respectively.

2.1.3.5 Apache Spark

Apache Spark is a modular data processing framework with Java, Scala and Python APIs [1]. Spark uses partitioned data sets, called resilient distributed data set (RDD), to provide fault tolerance over data sets that have been applied parallel computations such as map reduce or filter operations or a combination of operations. Apache Hadoop is used as the default distributed file system in Spark. The RDDs consists of partitions that are in Hadoop's file system the size of a Hadoop file block (64MB)[1]. RDDs are stored primary in-memory in each of the node, called worker, that operates an RDD partition. This allows to reuse the main memory to store intermediate results of the RDDs instead of writing the interphase results to disk. Thus, tasks that operate on data iteratively can gain a significant performance speedup by taking advantage of fast in-memory [66]. If the data set partition cannot be cached to in-memory, the worker writes it to the disk.

In Spark, driver is an entity which role is to run a job defined in application available to the driver and delegate the jobs tasks to worker nodes. Spark maintains a lineage history of transformations and actions that have been applied to the RDDs. A transformation is for example a map operation on the RDD. They are lazily evaluated, meaning that Spark does not perform computation for each transformation operation. On the other hand, RDD actions, such as count or map reduce operations block the execution of the driver program, which forces the driver to evaluate the operation by acquiring suitable worker nodes and sending the tasks to the workers.

Each driver program exists within a Spark context environment object. Drivers can be started as standalone using private context, or by sharing the context with several different driver programs. Operations provided by Spark's cluster manager are thread-safe which makes it available to service several drivers in concurrent manner. Thus, there are two options for starting Spark jobs against a Spark supported cluster. The first option is to start a new driver program within its own context every time a job is submitted. The second option, is to share the context object with many drivers. For the latter option, for example ooyla provides Spark as a service by implementing a job server to which the users can submit their own Spark driver programs through a REST interface [12].

Standalone scheduler is also called in Spark as master or cluster manager. Spark has support for several cluster managers. Standalone scheduler receives a registration request from the cluster workers and maintains a list of free worker nodes in a first in first out (FIFO) queue. Other options for resource management are Mesos and yet another resource negotiator (YARN). They

[1]https://spark.apache.org/docs/0.8.1/scala-programming-guide.html

differ from the standalone scheduler in terms of providing support for running different cluster computing frameworks.

Sparks flow starts when the driver application is started and it sends a notification of a forthcoming job to the cluster manager which selects suitable workers and communicates this information back to the driver. Once resources have been reserved for the job, the driver sends the application's code and tasks to the assigned workers.

A spark worker runs one or more executors in its dedicated environment. It is configured with a memory amount and number of cores for completing the tasks. Each worker may store the result RDD partitions into their cache. If a worker node is lost and with it the computed partition of an RDD, the lost partition can be recomputed from a transformation graph that originally were applied by the worker to create it. This provides fault tolerance.

Spark utilizes Akka framework [2] for abstracting communication details from the Spark application developers. The classes Standalone scheduler, Worker, Executor and Driver all extend class Actor of the Akka framework. Akka actors are entities that can send and receive messages to each other synchronously or asynchronously. The benefit of the actor model, is that it helps to manage concurrent management of threads and implements the communication protocols between the actor entities.

The author of the thesis in [51] measured an Asus Nexus 7 2012 energy usage and CPU and memory consumption for vision recognition and text processing applications in both offloading and local only computation modes. In the experiments the offloading was delegated to a cloudlet desktop computer using high-bandwidth and low latency WLAN network connection. Offloading with Apache Spark had significant performance advantage over local execution in terms of the tablet's energy usage and application's running time. Furthermore, the WLAN connection was disconnected at predetermined times before the offloading computation was ready. After this the computation was run only locally. The results showed that in cases when the WLAN connection was cut before 15 seconds has passed in offloadable computation, no performance gain was achieved from offloading for the reason that no single tasks had been completed in the cloudlet. In other cases, the tablet was able to reduce execution time and energy usage due to being able to use completed tasks results in the cloudlet decreasing the number of tasks it had to execute locally.

2.1.4 Applications

2.1.4.1 Dessy Indexer

Dessy indexer is targeted for mobile desktop search. The original application was ported to mobile phones using mobile information device profile (MIDP) 2.0 / connected limited device configuration (CLDC) 1.1 in [46]. Dessy indexes both file metadata and content. Photos tagged with exchangeable image file format (EXIF) location information are also supported. User of the application chooses the folder he/she wishes to synchronize between a mobile device and a remote device. Synchronization may be performed with both a server computer running in cloud service or another mobile device [47]. Synced documents may contain just the URL of the item if it is available for public downloading in Internet, thus reducing the need for data communication and energy usage of the mobile device.

Lagerspetz et al. summarize Dessy's mobile offloading energy usage in [47]. They measured energy consumption using WLAN (100 kB/s and 700 kB/s) and 3G (receive and transmit) with a Nokia N900@600MHz. They found that to transfer one megabyte of data via WLAN requires the task complexity to be 1 billion instruction in order to favor offloading. The experiment data set was compiled from Jane Austin's Pride and Prejudice and Grimm's fairy tales consisting of 181,692 words with average length of 4.59 characters and around one megabyte in file total size. The indexing component uses Syxaw [46, 50] which can also supports XML-file synchronization and includes support for efficient concurrent upstream and downstream synchronization using standard web protocols. The final results were that the offloading reduced energy usage by 97.6% for WLAN with 700 kB/s and 92.6% for WLAN with 100 kB/s.

2.1.4.2 Compiling

Chen et al. in [29] experimented dynamic application offloading to a resource-rich server for compiling Java bytecode to native code. They used Java's object serialization for transferring the migrated methods parameters to the server. Their framework can take into account the amount of necessary communication, network features, such as speed and type and energy required in compilation. The remote/local execution decision was implemented in the proxy for each component that may be offloaded. Experiments were conducted with four varying degrees of channel condition. They also tested with various sizes of input parameters. According to the results, when the poorest channel condition is prevalent and the input size is highest with an 512*512 image, local execution was favored over remote execution. With

higher bandwidth channels, offloading was optimal strategy in terms of energy consumption. For compiling experiments, which required only minor level of compiling optimizations, consumed less energy when executed in local mobile device regardless of the channel's condition. For more advanced compiling optimizations, energy was consumed less with offloading regardless of the channel condition.

2.1.4.3 Mobile Browsers

With the development of SPDY protocol by Google in 2009 as replacement for HTTP 1.0/1.1 protocol in clients and servers, several web browsers, such as Opera Mini [14], Nokia Xpress [11] and Amazon Silk [21], have decided to implement the protocol. The main design goals for SPDY is to reduce web page load time by decreasing the required two-way communication round trips between the server and client and by allowing request multiplexing, header compression and server push. Request multiplexing allows multiple request to be sent over one connection without waiting for the response before commencing a new request. Also prioritization of request streams are supported. In HTTP the client sends header information including state information stored in cookies over each request, thus transmitting redundant data. SPDY solves this by utilizing prior knowledge of common header information. Server push is another improvement in SPDY which allows the server to send a resource to a client without needing to wait a request for it. This is useful for example when the client loads an web page and an image is referenced from the web page. By sending the image immediately, the server decreases the number of round trips. [60]

The need for server side support of SPDY may not be needed to consider in case SPDY gateways are used as intermediate communication node between the client and server. SPDY gateways may be provided by commercial organizations or non-profit organizations [60]. For example, Amazon, Opera and Nokia provide SPDY gateway from which their respective browsers fetch web content. The gateway usually resides at a high-bandwidth Internet connection link and it may cache content.

In addition to reducing latency and the need of bandwidth with SPDY gateway, a mobile browser may decide which parts of a web page are executed at the client and which on the SPDY gateway's nearby servers. For example, Amazon offloads web page computation to the Amazon's S3 servers and Opera Mini to its Opera's cloud servers [64]. Processing parts of the web pages by applying cascading style sheet (CSS) rules or modifying the document with Javascript at the server side increases the amount of data that is transferred to the client. The increase in total size of transferred data is

offset by compressing the data before sending it to the client [64].

A cloud-aware system is designed and implemented in [24] for both supporting mobile browsing offloading and application offloading for Android platform. The authors conducted experiments by installing their system called CDroid (Cloud Android) to a set of user for two weeks. For the first week the users continued to use their phone in plain Android mode for taking control measurements. For the second week, CDroid was programmed to enable itself automatically directing all network communications through the CDroid server. First, the authors emphasize considerable savings the devices gained by reducing the amount of time spent in FACH power saving state over the day. Mobile phones remains in this half-power state after each time it has sent or received data using 3G. By setting up push notifications interval to 5 minutes, the study found that the phones consumed 5.3% of battery capacity per day while with plain Android the users phone spent 13.5% of battery capacity per day. Second, the authors state that the overall traffic (images+ads and data) received by the phone was reduced by 62%. According to the authors, this resulted from CDroid servers ads blocking mechanism and compression of web site content and pictures. In addition, caching at the server and optimized sync component helped in saving energy for user uploaded content. For example when a user uploads pictures to Facebook they are cached at the server and utilizing the sync module, CDroid only uploads new or modified content to the cloud. CDroid's synchronization module uses *inotify* to detect changed files and uses its own version of Rsync algorithm to update the source and destination directories [25].

In a second experiment, the authors of CDroid modified the phone's browser to take the time to load a web page, measure used energy and track amount of received and sent data. The experiment was carried in lab and included two web sites, m.9gag.com and m.facebook.com. The 9gag page's all day loading time was reduced by 5X and all day energy usage by 4X while Facebook's respective results were 2.7X and 2.25X. The former web page included more pictures and ads which explained the higher savings in terms of web page loading time and energy consumption compared to Facebook.

2.2 Energy Efficient Computing

In [37] the authors study the effect of thermal influence on processors energy efficiency using three different cooling systems while varying the workload intensity and repeat each experiment by changing the scheduling policy of allocating workload to different processor. In the first policy the workload is given to a single core, while in the second policy the workload is divided

equally among 4 processor cores. Their results are that with low workloads the consolidation scheduling policy is more energy efficient over the many core scheduling policy as in the latter case, the cores use extra power for static power dissipation even though their frequency and voltage levels are lower than used in the consolidation policy. With higher workloads, the many core policy becomes more favorable of the two methods, as its performance is higher and this results in better performance per watt than with consolidation.

Another performance consideration is to employ parallelism more efficiently in for example in face detection and face recognizing. This is motivated for example by the results of [37] who conclude that under high workloads it is more efficient to use several cores which consume more energy than on average in consolidation but which takes less time to complete. The use of graphics acceleration is studied for high performance computing (HPC) tasks for Exascale in [54]. The author evaluates and compares the performance and energy efficiency of Exascale architectures which are built using ARM Cortex A9 0.25W as low power processors, Intel Xeon E5 95W as high power processors and finally in the third case accelerators using NVIDIA GPU Tesla K20 225W. The benchmark used in the work was Rodinia Benchmark Suite of which three benchmark were selected which involved fluid dynamics computation, thermal simulation and optimization task for DNA sequency alignments. All benchmarks were written in programming model supported natively by each selected architecture. The author reports that the architecture using GPUs were 5 times faster and consumed 18 times less energy for all experiments.

In this Thesis, the mobile device used for conducting experiments was Asus Nexus 7 2013 tablet. It is equipped with Adreno 320 GPU and has support for Open Computing Language (OpenCL) 1.1 and Open Graphics Library (OpenGL) ES 3.0. The use of parallelism may result in total lower energy usage compared to using the tablet's central processing units (CPU) for face recognizing experiment.

2.3 Data Formats

Data that records relationships and hierarchies between stored elements are usually stored in relational databases, document databases or structured documents, such as extended markup language (XML). Storing such structured information in a space efficient manner and synchronizing it between the storage locations may require to use a set of specific methods.

In [49], Lindholm et al. present a method for performing XML differencing

and implement a tool which performance is measured in terms of running time for different data sets. In the test runs a binary diff tool outperformed all XML differencing tools and their own tool was one of the fastest among XML diff tools. In [50], the authors continue by presenting a synchronizer tool that supports XML file synchronizing and merging in addition to generic file synchronization.

For XML transfer and storage, the data may be stored in efficient XML interchange (EXI) format. In this format XML is converted into streams of events, which can either be structure or content events. This stream is further rearranged and compressed.

2.4 Update Detection

File directory synchronization starts from determining the list of changed files. This phase may be expensive for large directory trees. Instead of comparing the state of file directory trees, one alternative solution is to listen to file system events that the OS creates according to certain event filters and log these events in regular intervals. For example, *inotify* is a utility for Linux that supports monitoring file system events [7]. The users may add watches to file directories and specify filters to specify the set of file system event types he/she needs to track. The events are read from a file descriptor. Authors in [24] reported that their logging implementation using *inotify* didn't have significant impact to user experience in terms of performance.

Chapter 3

Data Synchronization Methods

This chapter discusses data synchronization methods for performing one-way and two-way synchronization. In addition, incremental backup can be used in case the mobile device stores the master copy. Three incremental backup systems are presented, one which is a application of a one-way synchronization utility, Rsync, one a recently developed file system, and finally a backup system for Apple's OS. In this Thesis, we have chosen to exclude database synchronization methods, such as designed in open source SymmetricDS[1].

3.1 Git

Git is a distributed version control system that supports multiple users to work with their own local repository. The users may work with their own version of the repository in their local environment and store a snapshot of their local file storage stage to the repository as a commit which they can at any point of time publish to other users. A server stores the shared repository, from which each customer may fetch the latest changes published by other users. The development of Git was started in 2005 by Linus Torvalds for Linux kernel development. One of the main motivation for the development work was to improve synchronization performance of other distributed version control systems. At the time of writing this Thesis Git has become one of the most used software configuration system. In this section we will focus on introducing Git's work flow and internal working with respect to data synchronization.

Susan Potter in [56] and John Wigley in [65] discuss the components used in Git's work flow. The working model consists of three components each

[1]http://www.symmetricds.org. Accessed: 23th January 2015

providing input to the next one in a cyclic manner. Git's work stages are described in Figure 3.1.

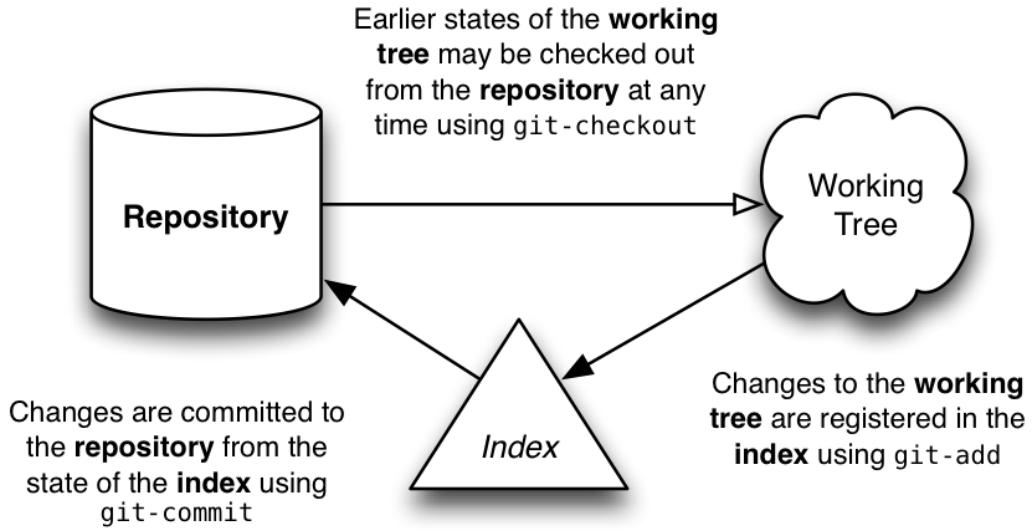


Figure 3.1: Git repository's work process as in [65].

The first component is working directory into which the user can checkout a snapshot of the repository called a commit or simply start adding files to a fresh empty working directory. The root of this directory includes a *.git*-folder which is used for maintaining the history of the repository including the log messages and authors of the commits.

As soon as a user modifies an existing file in the repository or adds a new file to the working directory, the user can move this file to a staged area, also called the index. The purpose of the staged area is to enable to make incrementally changes to the files in a step-by-step way that can be rapidly tested before actually deciding to store the changes into the repository. At the time of making a commit, only the files added to the staging area will be included in the new commit. This new commit will be added to repository's commit tree as its root node called HEAD.

Commit can become a children of another commit made at an earlier point of time. When two users modify files in their local working tree and publish their changes to shared repository a synchronization operation is required by other customers. Git comes with a push operation for transferring new commits to another repository and a pull operation to perform synchronization of the local repository with another repository. If two or more users have modified the same files, a merge operation will be required. In case the changes are made to same sections in the files and they differ from each other

a conflict situation occurs. This situation requires manually deciding which changes are kept in the final version.

Internally Git manages files as objects in the repository [28, 65]. An object can belong to one of the following classes: blob, tree, tag or commit. File content is stored into a single blob object. These are created as a user stages a new or modified file. Files with the same content are always mapped to the same hash identifier. A tree object can refer to several file objects and other tree objects. Thus, it functions similar to a file system directory. In contrast to file objects, commit object, depends on the time of creation, thus making two different users' identical commits still required to be merged during synchronization of the repository. Finally, tag object can refer to a previously made commit. An example Git repository is given in Figure 3.2.

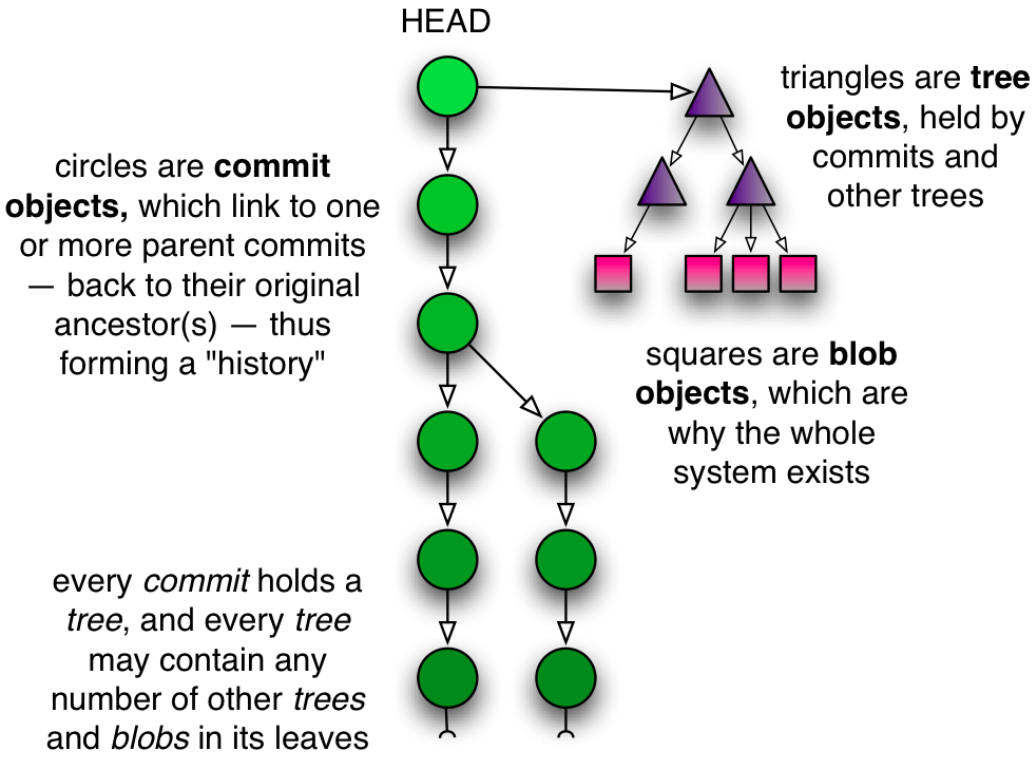


Figure 3.2: Git repository as in [65].

Git keeps track of the file history by recording the changes made to a file between each successive commits. Files that do not change between two commits are referenced by their hash identifier. Git uses the longest common subsequence method for finding the changes. When synchronizing the local and remote repositories, only the delta needs to be transferred as Git records for each delta object the previous delta object or the base file. Reconstruction

of a file's specific version may require several iterations.

Each time a user adds the same file to the staged area, Git creates a new blob. This can become inefficient with respect to synchronization and storage size. Thus, Git can periodically form pack files which collect related files and store the changes of the files as delta compressed.

3.2 Rsync

Rsync was developed in 1999 by Andrew Tridgell for remote data update on the basis of zlib [22]. In [62], Tridgell justifies the need for developing a data synchronization method which minimizes the amount of transferred data due to restriction in available network bandwidth. He also states that network transfer costs dominate the local file system access time.

We use the term version string to denote a file at the local side, and the term reference string to denote a file at the remote side. The aim is to transfer only the compressed delta between the local and remote endpoints. The delta includes position information to add or remove characters in relation to the reference string.

Both the version and reference strings are divided into equal sized blocks and the blocks from the reference string are compared with those of the version string. To detect changes at block level a summary value of the block, called fingerprint or signature, is computed for each of the block at the remote endpoint. Fingerprint calculation can take major time of the algorithm's running time. This has been solved by using two level fingerprinting: The first level fingerprint is obtained by using computationally cheap algorithm, while the second level fingerprint is calculated with a cryptographically strong algorithm. Rsync uses MD5 for computing the strong fingerprint [61] which makes the probability of fingerprint collision between two different blocks very low [62].

We call the party which wants to publish its changes as A the sender and the party which wants to update its destination files as B the receiver. At the beginning A needs to know which files have changed since the last synchronization round. When the folder tree is large and the number of changes is relatively low, sending the file list may dominate the running time. This is solved by forming the file list at both A and B and using Rsync's algorithm to update the file list at A. Then only files whose time stamp or size has changed will need to be updated.

```
1 for each file_id received from Generator
```

```
2 A receives weak and strong block signatures for the
   ↪ file id from B
3 A construct a hash table using the weak signature as
   ↪ an index (first 16 bit)
loop:
4 A computes rolling hash sum for its file's current
   ↪ block
5 A looks up the signature from the hash table
6 if (hash_table[A.block.signature] != null)
7  then
8    emit block found to Generator
9    change the file reading head to the next block
10 else
11   emit the current character in the file to the
   ↪ Generator
12  change the file's reading head by one character
   ↪ forward

13 fi
14 loop until each block in the version string has been
   ↪  checked
```

Listing 1: Rsync synchronization algorithm

To begin the file synchronization B must compute for each block both the cheap and strong fingerprint. This data is then send to A which creates an index table and a signature table from the received fingerprints. The index table is indexed by the first 16 bit of the cheap fingerprint allowing to quickly look up an entry in the signature table which contains the full cheap signature and strong signature.

Listing 1 illustrates the Rsync algorithm at the sender side. When an entry is found in the signature table, A computes the strong signature for the block. If they match, the blocks in the reference and version string are the same, and A sends a match to B and advances to inspect the next block in the version string. Otherwise the blocks differ, and A adds one byte from the version string to the file's list of changed characters and advances the read head by one byte. Next, A computes cheap fingerprint, also called rolling hash, for the current block wide part of the version string and consults the index table for a matching entry in the signature table. Tridgell states [62] that any algorithm with rolling property may be used for computing the rolling hash. An algorithm with this property can compute the rolling hash

when the buffer is transferred by one byte. Tridgell continues that the choice of the fast signature algorithm is a trade-off between speed and effective bit strength.

The rolling hash technique is used to detect transposed substrings in the version string. For example, inserted text in the beginning of a file shifts the contents of each block of the file, which changes the blocks' signatures. With the use of index table and signature tables, Rsync finds transposed blocks in constant time bound.

Delta is the change between the version string and the reference string. It includes position information for the receiver to add or remove characters in relation to the reference string and index data for common blocks. The change is compressed by utilizing the context of shared blocks between the reference string and the version string. Rsync compression method achieved in [62] 1-6% improved compression ratio for delta compression when the deflate algorithm used in gzip was complemented with context information. At the time of writing this Thesis, Rsync uses zlib for compression and utilizes the shared content in matching blocks to achieve improved compression ratio.

The block size affects to how much fingerprint data is sent from B to A. The condition that must be met is that the block size must be large enough so that the amount of transferred data is larger than the amount of transferred fingerprint data. The benefit of smaller blocks is that they allow to detect local changes if the changes are small but uniformly dispersed over the file. Rsync selects the block size for each file on reference of the file's size.

Latency may become a dominating factor when synchronization is run for endpoints connected by a network. Thus, Rsync allows to pipeline the synchronization of files. Then the sender does not need to wait to start a second file synchronization before the first file's synchronization has finished [62]. The actors and message flows in the synchronization process is given in Figure 3.3. Notifications are sent by the receiver to the generator when a file has not been successfully received. In this case, the generator creates larger per-block fingerprints and starts the file synchronization again.

3.3 Incremental Backup

3.3.1 Rdiff-backup

Utilizing the same synchronization algorithm as Rsync, Rdiff-backup creates an incremental backup from the sender's file directory to the receiver's file system with each synchronization run [26]. The sender and the receiver must both run Rdiff-backup clients. The sender's role is to compute the deltas

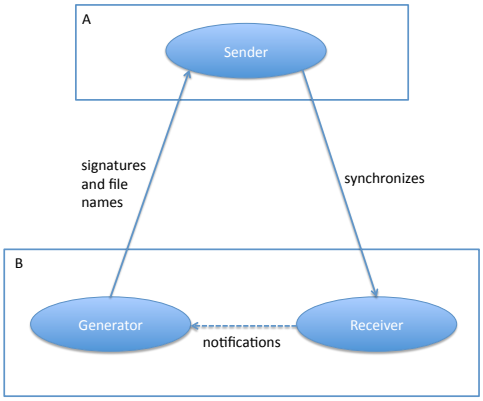


Figure 3.3: Rsync synchronization process' actors.

over the synchronized files and send them over to the receiver using the same protocol as in Rsync. The receiver reconstructs each of its files using the transmitted deltas and also computes the reverse of the delta. In order to restore a specific version of a file, Rdiff-backup applies the file's incremental backups in successive order to the most recent snapshot of the file. Space efficiency is favored over speed of computing the file's specific snapshot.

3.3.2 Btrfs

Btrfs is a copy-on-write (CoW) file system developed for Linux by multiple companies and licensed under GNU General Public License (GPL) [6]. It is supported by most recently distributed kernel versions (from 3.17 distribution). One-way synchronization can be performed by using incremental backups. First, the user creates a snapshot of the synchronized folder and upon each synchronization time point creates a new snapshot of the folder and sends only the changed file blocks between the most recent and previous time points. The backup machine must run an instance of a Btrfs server to receive the initial full snapshot of the synchronized folder and the subsequent incremental backups.

Incremental backups utilize copy-on-write feature to create rapidly snapshots of the synchronized file directory as a snapshot in Btrfs contains only links to the files of the original file system directory. Each created snapshot is written to a specified file directory as a subvolume. Subvolumes can be created as read-only or writable and they can be either mounted and un-

mounted. Once a file is changed in the original directory, Btrfs creates a copy of it. Thus, the snapshot's state remains intact while the file system updates the changed file's block list to include the references to the new and updated blocks. When sending an incremental backup, Btrfs determines the changed blocks between the two snapshots and completes the process with Btrfs send at the synchronizer machine and Btrfs receive at the backup host machine [6, 52].

3.3.3 Apple Time Machine

Apple Time Machine is an incremental backup software which stores snapshots of the file system and allows to browse the file system as it appeared on the date of creating the snapshot [8]. Initially the software copies the directory tree from all mounted disks to a Time Machine volume, except for user specified files. On subsequent backups, only the changed files needs to be transferred. Changes are tracked by directory and the backup software does not need to scan all changed files. These modifications are logged by Apple system event service. For files that have not changed, a hard link is created to point to the existing copy of the file on the backup volume.

Chapter 4

Experiments

This chapter presents the results from data synchronization tests. We begin by describing the test setup and the chosen test applications. After this we review the measured quantities in the tests and lastly, represent the results for the chosen data synchronization methods and applications.

4.1 Test Setup

The general layout of the experiment setup follows closely the experiments conducted in [51]. Most of the tests were run on Asus Nexus 2013 tablet, which was monitored through an secure shell (SSH) connection. Before each test was started, the universal serial bus (USB) cable connecting the tablet and desktop machine used for monitoring and offloading computation was unplugged from the tablet.

4.1.1 Hardware

As a mobile client we used initially Asus Nexus 7 2012 quad-core 1.2 GHz tablet with NVIDIA Tegra 3 and 1GB of main memory. Later, we decided to continue testing with the updated model of Asus Nexus 7 2013 quad-core 1.5 GHz tablet with Adreno 320 and 2GB of main memory. Both client devices support WLAN 802.11g. The cloudlet machine in the experiments was a Dell Optiplex Intel Quad CPU Q9400@2.66GHz desktop computer with 8GB of main memory.

The tablet and the cloudlet communicated via a Linksys wireless access point router. The tablet used WLAN connection to the router while the desktop machine was connected via an ethernet cable to the router. To administer and perform test configurations and test launches, the cloudlet

used an USB cable to connect to the tablet. Android debug bridge (adb) was used to issue manual commands to the tablet. Once the tests were started, the USB cable was disconnected from the tablet for collecting proper power readings during the test runs.

4.1.2 Software

Asus Nexus 7 2012 tablet used Ubuntu Touch [19] 13 OS as it provided support to run Spark applications which required software not available for Android 4.1 OS [51]. Ubuntu Touch 14 OS was initially installed on the Nexus 7 2013 tablet. Finally, the tablet's OS was updated to Ubuntu Touch 15.04[1] (version 24). The software stack for both tablets consisted of Java 1.7 (with update 45) for ARM processor architecture, Spark 0.8 [1], Akka framework 2.0.5 [2], Scala 2.9.3 [15] and image processing library Open Source Computer Vision (OpenCV) 2.4.6 [13]. In addition, the Spark applications used Hadoop 1.04 [3] for reading test data.

Modified versions of Spark and Akka frameworks were used in order ensure that the tablet can continue computing with a local fail-safe worker in case the network connection is lost. [51]. Unix *redir*-utility was used for redirection of network connections between the tablet's external worker and the tablet's Spark master and Spark driver [9]. In addition, the Spark framework had been modified by introducing a new configuration variable to the Spark worker by allowing to pass it a chosen priority value upon start of the worker node. The Spark cluster manager chooses workers for a given job on the basis of the workers priority value before it compares the available resources between the workers [51].

Xubuntu 12 using Linux 3.11 was used as the desktop machine's OS. The software stack consisting of Java, Spark and Scala was also installed for the desktop cloudlet to support Spark applications. Differences between the cloudlet and tablet software were that the cloudlet used newer version of Java 1.7 (with update 67) and also newer version of OpenCV (2.4.9).

4.1.2.1 Git

Git 2.1.3 was used in the tablet, while the desktop machine used Git 1.7.9. Both machines' repositories used default settings. Before proceeding to the data synchronization phase, a bare shared Git repository was initialised to the desktop server. Secondly, in the client side, an empty repository was initialised for each of the tests. SSH was used to encrypt the communication

[1]http://news.softpedia.com/news/Ubuntu-Touch-15-04-Vivid-Vervet-Is-Open-for-Business-First-Image-Is-Out-463572.shtml. Accessed 25th January 2015.

between the two synchronized machines. The authentication utilized public and private RSA key pairs.

The measured quantities included the phase for adding the data objects to the local repository, the following commit to the repository and finally the transfer of data files to the desktop machine. Initially garbage collection (*git gc*) was executed just before starting the file transfer. This phase optimizes the repository's size by removing unreachable objects and by generating a compressed binary archive, called the pack file, reflecting the current state of the repository. It stores the changes applied to each repository object for its entire history in compressed format. Any given revision of a repository's file can then be reconstructed by applying the deltas successively to the most recent snapshot stored in the repository. While garbage collection does optimize the size of the repository, it does has an effect on performance. With *git gc* we observed increase in running time and energy usage of the synchronization tests. Git runs this phase automatically when the number of loose objects (blobs not stored in a pack file) reaches a threshold value. As Git also computes and compresses the deltas before commencing the file transfer, we decided to not include the optimization phase.

4.1.2.2 Rsync

Rsync 3.1.1 was chosen as the second data synchronization tool. File transfer used Rsync's protocol, which, like Git, uses SSH for creating a secure communication channel between the two synchronized computation nodes. Rsync was configured to use archive mode, which preserves symbolic links, attributes, permissions etc. and additionally, compresses the files. For the third class of tests in 4.1.4, we determined the changed files by using Rsync's option for sending the file list to the tablet without actually starting the data synchronization.

4.1.3 Measured Quantities

In addition to test running time, we measured three primary quantities during the tests. These were CPU and memory usage and power consumption. CPU and memory readings were obtained with the Unix *ps*-utility. UPower 0.9.23 was used for collecting power measurements in Nexus 2012. Voltage and current values were obtained directly from */sys/class/power_supply* folder when estimating power usage in Nexus 2013. Power is directly proportional to the level of current and voltage at a given measurement time point. The device reports current level in microamperes and voltage level in millivolts.

The measurement readings were collected once in a second. Most of the tests launched several synchronization process instances. Thus, it was necessary to record a time step value for each instance and to compute the aggregated measurement values for each time step.

Nexus 2013 battery is designed to last for approximately 9 hours and its capacity is 3,950 mAh [5]. The device includes a micro controller, which reports the current and voltage levels [17]. As the level of total charge drops, the power estimation computed based on the battery gauge decreases compared to taking direct power measurement values using voltage and current meters. Thus, the device was charged fully before starting up test runs.

In addition to the measurement quantities that were collected for each test, we recorded the amount of time required in different synchronization phases. For Git, we measured the time to add the changes from the local file working tree into the Git's data repository, the time to make commit, and finally the time to transfer the data from the tablet over to the cloudlet repository. In Nexus 2013 Git tests did not optimize the repository by performing garbage collection, while this phase was included in Nexus 2012.

The power measurement output included an energy rate field in Nexus 2012 which was used to estimate the total energy consumption of the tablet during the test runs. The energy for each test was obtained by integrating power usage over the test time. We made an assumption that the power usage remains constant over the time until the next power measurement time point. The timing interval was set to one second for offloading and data synchronization tests and to 0.5 second for file list building tests. In practice, the spacing between time measurements points were higher than the intended interval (0.5 - 1Hz).

4.1.4 Tests

In this study it is assumed that the mobile client is not attached to an external power source for most part of the day and the client must rely on batteries. For this reason the main comparison between the synchronization methods is examined based on power consumption which is used for computing the tablet's total energy consumption.

Nexus 2013 device was observed to go into suspension mode after a few minutes of inactivity period with no direct user input. This caused problems related to completing test runs without offloading when the Spark driver program repeatedly reported timeouts and tried to re register the BlockManager of the local Spark worker. Additionally, data set synchronization tests were observed to stop progressing for several seconds. After this it was decided to issue an active request before starting each test with *powerd-cli* tool [20].

This reduced overall running time for each test case. In addition, variability in power levels were reduced significantly.

The tests in this study are divided into three classes. In the first class of tests, the execution cost of computation intensive application in addition to data synchronization costs were measured. Tests belonging to the second class computed locally with new files in the tablet and computed remotely with files transferred at an earlier time point. The experiments included the time for waiting the offloaded computation to finish and the receiving of the results. Furthermore, we measured separately the running time and energy cost for determining the changed files. Finally, in the third class, the purpose was to compare the applicability of the chosen data synchronization methods with respect to measured quantities. These tests included test runs which synchronized file trees consisting of both only new files and updated as well as new files.

In the first test category, the chosen test application was recognizing a person's face from Caltech image data set consisting of 449 pictures [4]. The face recognition experiment included tests with variable amount of transferred images starting from full synchronization of the picture database and a 20% reduction to the image database in the subsequent tests until no data synchronization was required. In addition, test cases were executed more frequently between 80% and 100% with 5% stepping in order to estimate variability of time usage when using partial offloading.

Both the tablet's and the cloudlet's Spark workers were configured to use maximum of three CPU cores and 384 MB of main memory. When testing with higher amounts of Spark worker RAM, Spark driver reported that the worker had not accepted any jobs. This occurs when a given worker starts a new job and is not able to reserve the amount of resources given in the passed configuration variables.

Temperature's effect on energy efficiency was also evaluated by choosing one of the partial offloading experiment in which the tablet's worker was configured to use two CPUs in addition to running the same test with three cores. Finally, the 20-80% partial offloading experiments were run also with suspension support. One of these tests was conducted with Ubuntu Touch 14 OS, while the other tests used Ubuntu Touch 15.

For the third category of tests, we selected full synchronization of Linux 2.6.32.63, its update to Linux 3.15.1 and the previous version's update to a closer version (3.17). The two latter experiments also tested the synchronization methods performance for constructing the changed file list.

Before the start of each test execution care was taken to close any unnecessary applications running in the background. The tablet was not used for a full minute before launching test runs to ensure the screen were not drawing

any power. Certain test runs were monitored by inspecting log files using SSH session between the cloudlet and the tablet. Most of the offloading tests were run twice, except for the with offloading with suspension test as well as 20-60% partial offloading tests with suspension which were run once. The data synchronization tests were also measured once and finally, the file list building tests three times.

4.2 Caltech Data Set Synchronization

Caltech data set consisting of frontal face pictures [4] was used to perform various synchronization tests by varying the percentage of pictures as shown in Table 4.1. Git uses by default compression, while Rsync does not compress JPEG files[2]. Compression level can be changed in Git, but this affects all files stored into the repository.

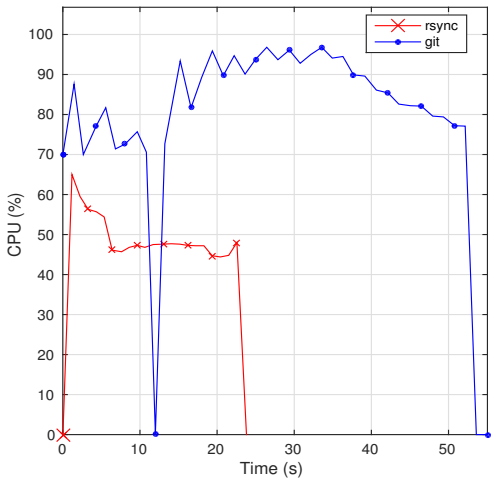
Table 4.1: The synchronization tests conducted with the Caltech dataset.

Test by percentage of pictures	5	10	15	20	40	60	80	100
Count	22	45	67	90	180	269	359	449
Size (MiB)	3.7	7.2	10.5	14.3	27.4	41.5	57.5	71.2
Size (%)	5.2	10.1	14.8	20.1	38.5	58.4	80.9	100

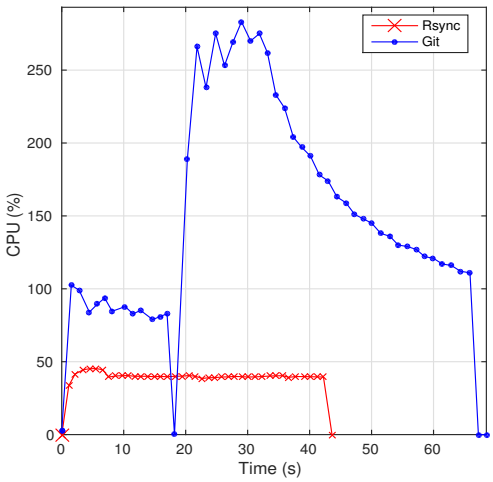
Recompression of pictures lead to high resource usage rate for synchronizing the data set with Git, which utilized several CPUs for creating the pack file for the new objects in most of the tests in the beginning of *git push*-phase. Creating the pack file involves both compression and integrity checking. In one of the test (60%), Git used only one CPU as presented in Figure 4.1(a).

Rsync was less CPU intensive than Git as it does not compress the pictures, although it computes checksums for each file. This is shown for example in Figure 4.1(b). On the other hand, Figure 4.2(b) shows that during the full transfer test Rsync used approximately the same amount of power during the period of time when Git transferred the pictures (35-65 s) after creating the pack file. Finally, Figure 4.2(a) illustrates memory usage in one

[2]http://rsync.samba.org/ftp/rsync/rsync.html



(a) 60% data set synchronization CPU consumption with both Git and Rsync using 1 core.



(b) 100% data set synchronization CPU consumption with Git using 3 cores and Rsync 1 core.

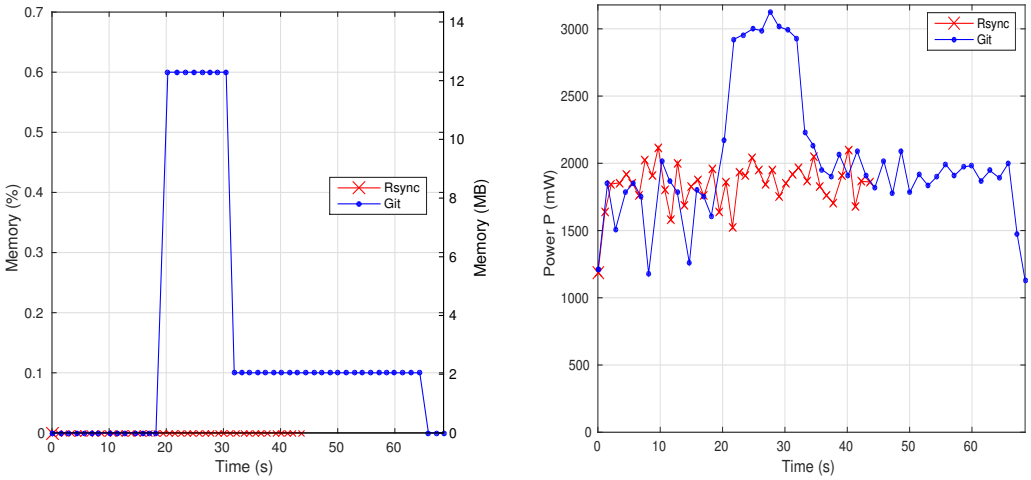
Figure 4.1: Asus Nexus 7 2013 60% and 100% Caltech data set synchronization CPU consumption, n = 449.

synchronization run for the 100% test. Git's memory requirement was highest during creating the pack file before the beginning of file transfer, while Rsync's memory usage remained low as it transfers the files one at a time.

The synchronization tests' running times are presented in Figure 4.3(b). Git used one core in the 60% test, which increased this test's running time and energy consumption compared to other tests that utilize several cores as seen from Figure 4.3(a) and Figure 4.4(a). Git used memory proportionally to the size of the synchronized data set as illustrated in Figure 4.4(b), while Rsync did not use measurable amount of memory in most of the tests.

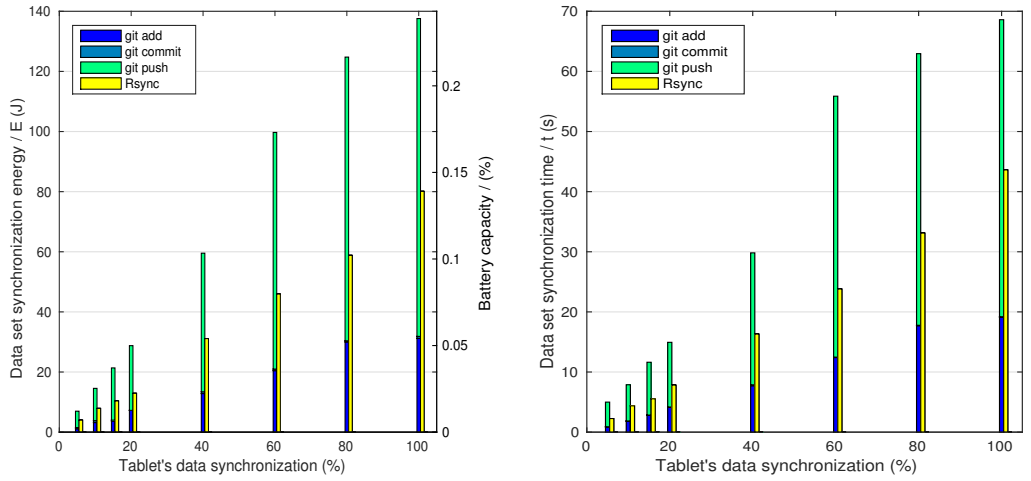
4.3 Face Recognizer

The face recognizer application used in this Thesis was developed in [51]. The picture selected as the reference image was preprocessed offline by using principal component analysis (PCA) to compute the principal components, also called as eigenvectors and eigenfaces, for the part of the picture containing the face. PCA transforms target data set space into a reduced orthogonal space by finding the most informative features describing the data set. It can be used for linear data. The precomputed eigenvectors were stored to the tablet. Before starting the recognition task, the driver application was



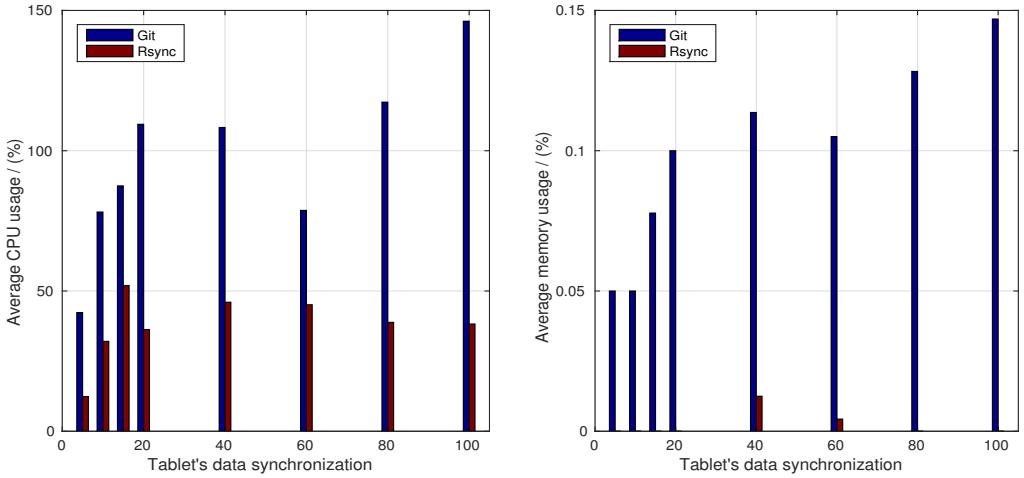
(a) Memory consumption. (b) Power usage.

Figure 4.2: Asus Nexus 7 2013 100% Caltech data set synchronization memory and power usage.



(a) Synchronization energy usage. (b) Synchronization running time.

Figure 4.3: Face recognizer data synchronization energy and time usage in Asus Nexus 7 2013. Git tests were run without optimization.



(a) Synchronization CPU usage.

(b) Synchronization memory usage.

Figure 4.4: Face recognizer data synchronization average CPU and memory usage in Asus Nexus 7 2013. Git tests were run without optimization.

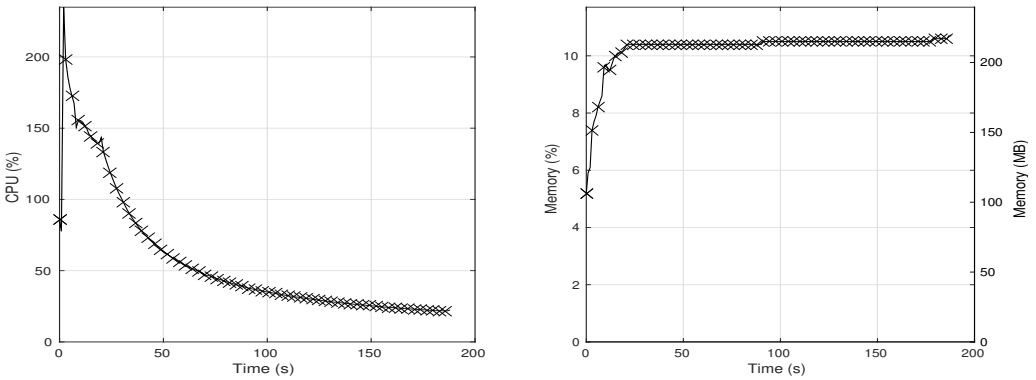
instructed to broadcast this shared data to each Spark worker node.

Face recognizer application consists of face detection and face recognition phases. Both phases used implementations of the algorithms from OpenCV library. In the first phase frontal faces were detected by gray-scaling the picture and then detecting the faces with a cascade classifier using Haar-like features. This algorithm is based on the work by Viola and Jones in [63]. The second phase applied the principal components obtained from the training picture to each pixel in the detected face area to project it to a reduced eigenvector space. In this reduced space the application computed distance between the training image and test image. The actual face recognition decision was performed by the driver application by comparing the distance to a predetermined threshold value [51].

Each picture in the Caltech image collection is 896 x 592 pixels and the faces detected by OpenCV were scaled to 96 x 96 pixels. The application found first 12 images of the referenced person without false detections in [51]. To find the remaining pictures the threshold value had to be increased, which also increased the number of false positives by 15 picture [51].

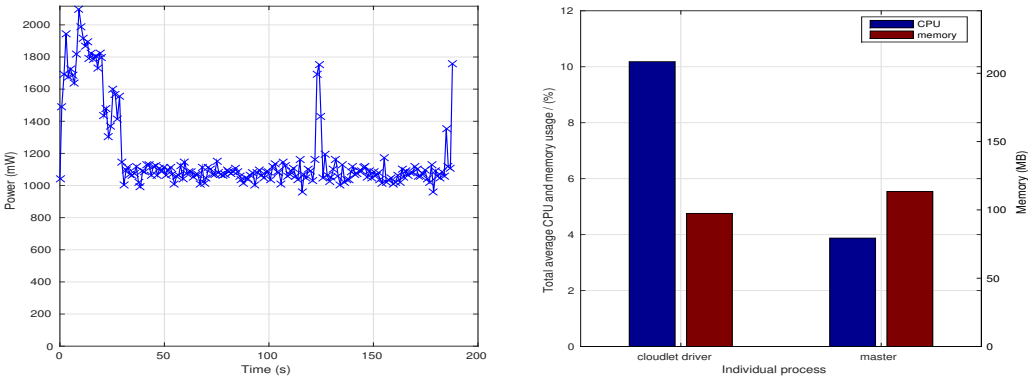
4.3.1 Offloading Face Recognition

Figures 4.5(a), 4.5(b) and 4.6(a) show resource usage levels for the offloading experiment. The highest resource needs are at the start of the test. The level of CPU usage steadily decreased as the test run progressed. With



(a) CPU consumption. (b) Memory consumption.

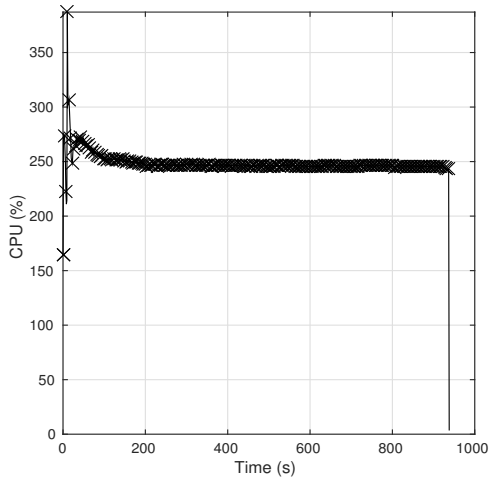
Figure 4.5: Tablet's memory and CPU expenditure with full offloading.



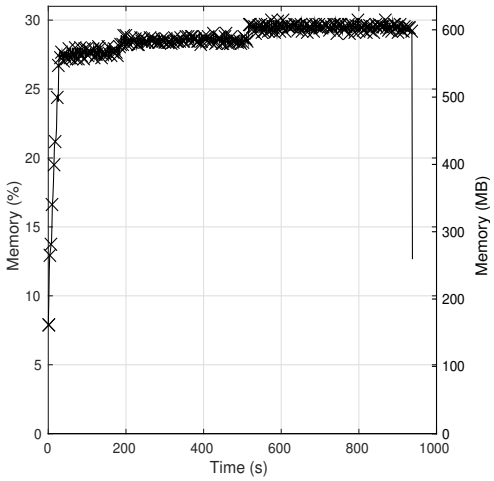
(a) Power usage. (b) Individual processes.

Figure 4.6: Tablet's power usage and individual process resource consumption with full offloading (n = 449).

offloading both the Spark master and the Spark driver used relatively more CPU compared to without offloading test as shown in Figure 4.6(b) and Figure 4.8(b). For comparision, the respective CPU, memory and power and individual process resource usage levels for the without offloading experiment are illustrated in Figures 4.7(a), 4.7(b) and 4.8(a).

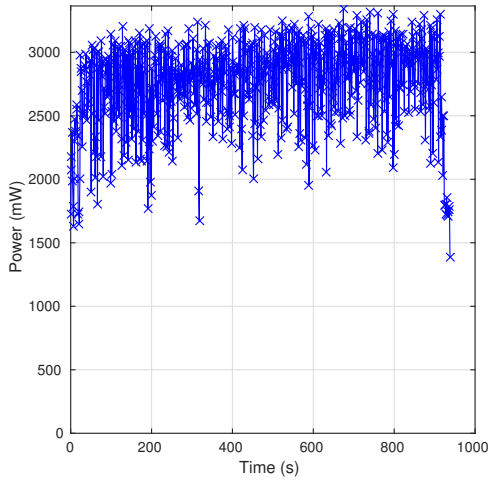


(a) CPU consumption.

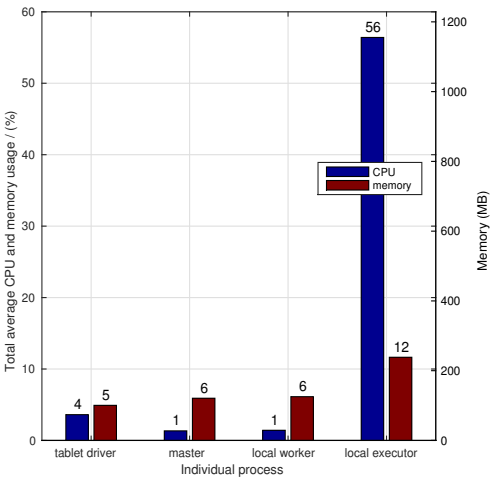


(b) Memory consumption.

Figure 4.7: Without offloading (n=449) CPU and memory consumption.

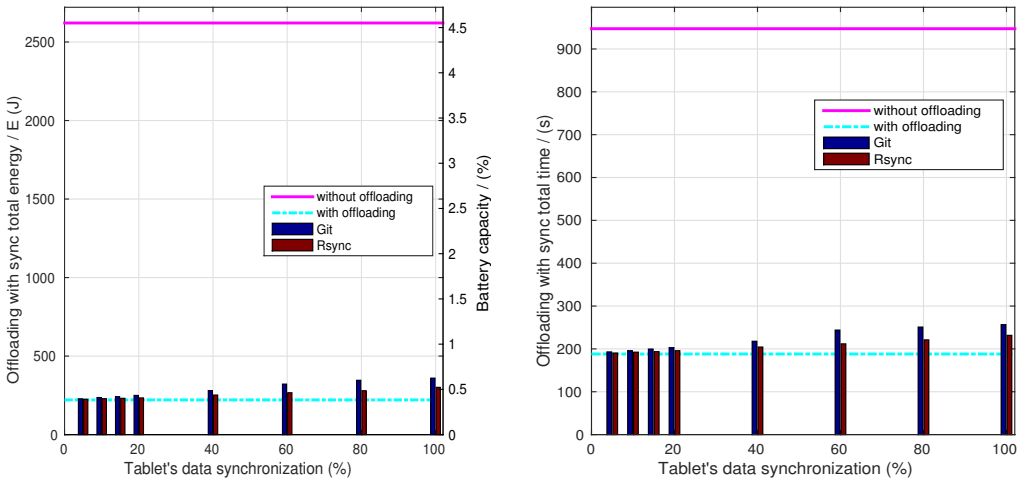


(a) Power usage.



(b) Individual processes.

Figure 4.8: Without offloading (n=449) power usage and individual process resource consumption.



(a) Total energy usage.

(b) Total running time.

Figure 4.9: Face recognizer offloading with data synchronization total energy and total time usage in Asus Nexus 7 2013. Git tests were run without optimization.

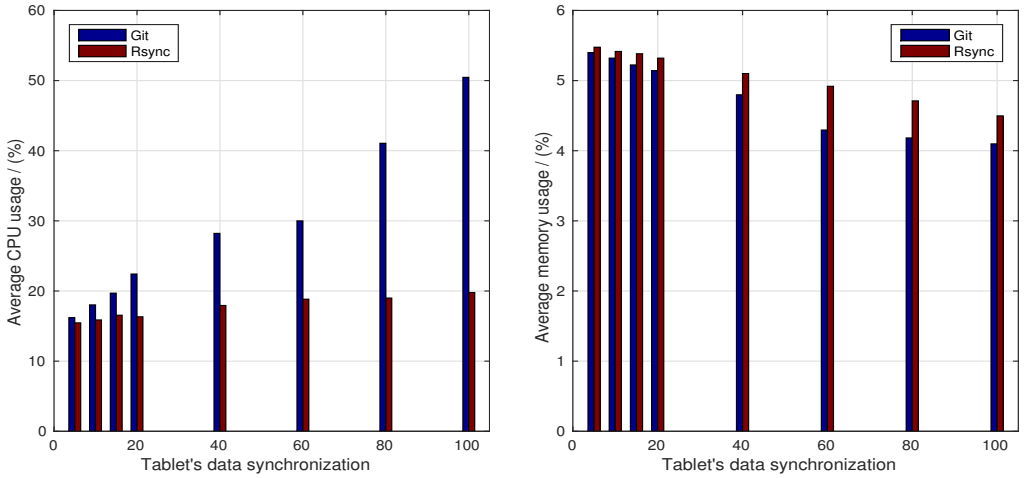
4.3.2 Offloading With Data Set Synchronization

Figure 4.9(a) shows the application's total energy expenditure for offloading with synchronization in Nexus 2013. Figure 4.9(b) summarizes the respective running times. Git used approximately 0.24% of the battery capacity in the 100% synchronization test while Rsync 0.14% as shown in Figure 4.3(a). Total energy usage without offloading with the whole data set on the tablet consumed around 4.6% of the total battery capacity while the 100% synchronization with offloading using Git consumed 0.65% and Rsync 0.55% of the total battery capacity. Finally, offloading with synchronization used CPU proportionally to the size of the transferred data set as shown in Figure 4.10(a), while consuming on average less memory proportionally to the size of the data set as Figure 4.10(b) illustrates.

4.3.3 Partial Offloading

The time for running partial offloading includes the time to find out which files have been changed and added since the previous synchronization round. Figure 4.11 illustrates this for Git and Rsync. Scheduling and utilization of disk cache may have affected the results for both methods. Git was used in the partial offloading tests to find out the changed files.

An additional approach for detecting changed files is to maintain the state



(a) Offloading with synchronization average CPU usage.

(b) Offloading with synchronization average memory usage.

Figure 4.10: Face recognizer offloading with data synchronization average CPU and memory usage in Asus Nexus 7 2013. The trend of average memory usage is decreasing as the synchronization uses less memory compared to offloading while taking a higher proportion of total task time. Git tests were run without optimization.

of the application's data directory for each synchronization round. When a partial offloading is commenced the application needs to construct the state of this folder and compare it to the state of the previous synchronization round using for example *diff*-tool. This approach was tested with this application and its running time was similar to Git.

We conducted partial offloading experiments using Asus Nexus 7 2013. Figure 4.12(a) presents the energy consumption of the tablet and Figure 4.12(b) the respective benefit factors when running face recognition in parallel at the client tablet and server. Figure 4.13(a) and Figure 4.13(b) show the experiments' running times and benefit factors respectively. The tests were conducted by running two instances of Spark driver programs which both connected to the same Spark master. The aim was to allow the cloudlet driver to select first the remote Spark worker. In order to ensure that the tablet driver did not choose the remote worker, a wait delay of 5 second was used between the launch of the driver programs. The driver programs were passed an image file name for the old files at the server or for the new files on the tablet.

Resource consumption in terms of CPU, memory and power for 60% partial offloading test case run is shown in 4.14(a), Figure 4.14(b) and Fig-

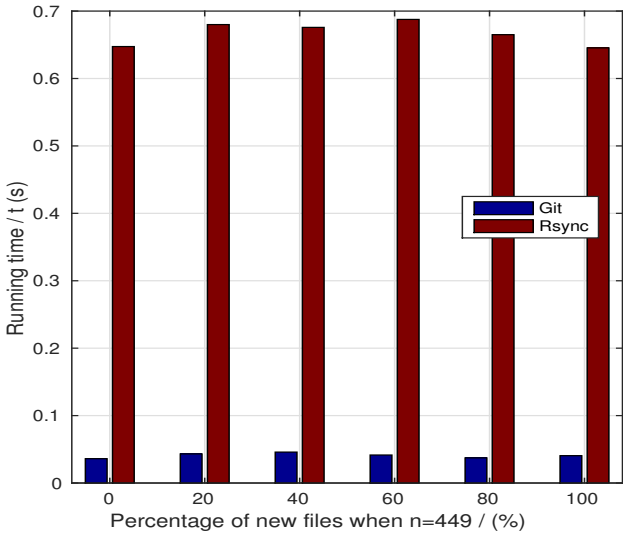
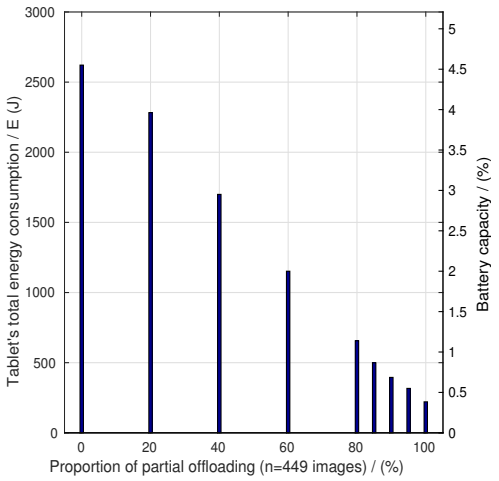
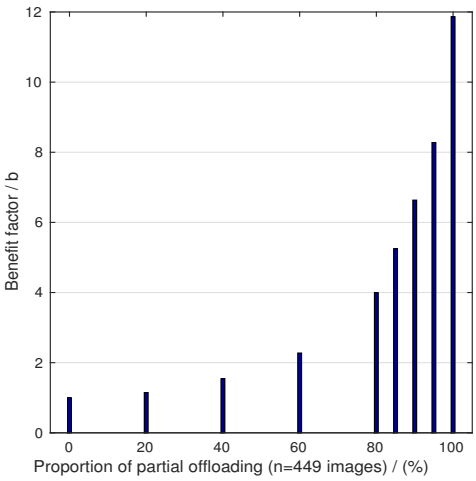


Figure 4.11: Face recognizer - times to build a list of changed files. Both test sets were run on Asus Nexus 7 2013 tablet.

ure 4.15(a). Figure 4.15(b) illustrates average individual process CPU usage and memory usage during the tests. In the partial offloading tests the cloudlet driver utilized significantly higher amount of CPU cycles than the local driver. Power usage levels had a higher variability as the load of the system increased which can be seen from the power usage Figures for the without offloading in 4.8(a), 60% partial offloading in 4.15(a) and with offloading tests in 4.6(a). The variability in power usage may be caused when the local executor issues a file system read operation for an image file from the tablet's flash memory, and remains idle while waiting for the file to be loaded into the main memory.

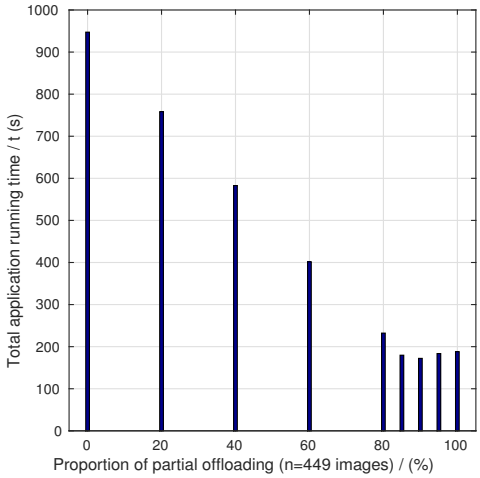


(a) Partial offloading energy consumption.

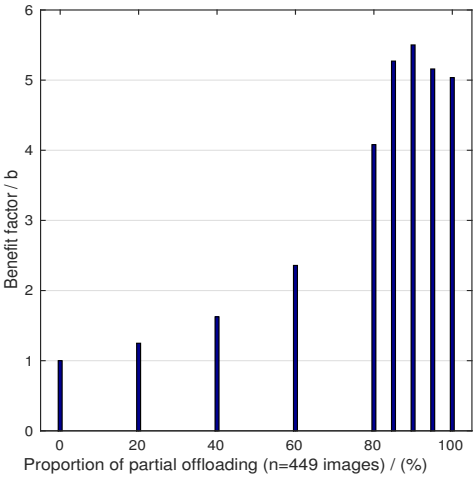


(b) Partial offloading benefit factors in terms of energy consumption.

Figure 4.12: Tablet's face recognition partial offloading energy consumption. Change detection was performed with Git.

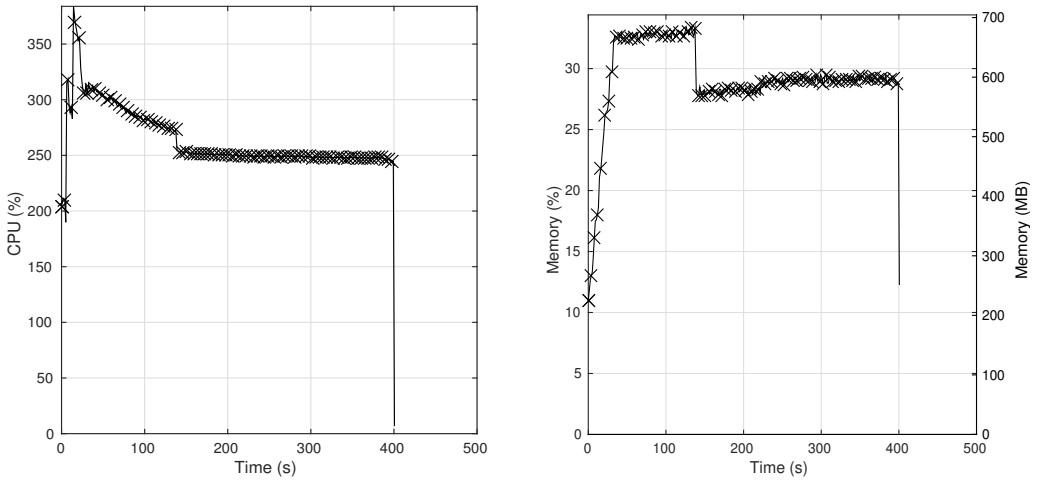


(a) Partial offloading running time.



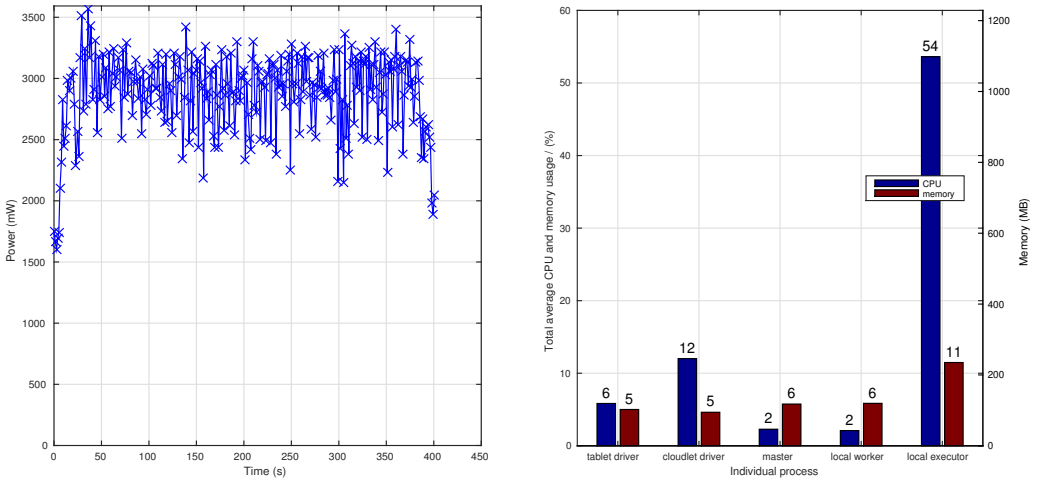
(b) Partial offloading benefit factors in terms of running time.

Figure 4.13: Tablet's face recognition partial offloading running time. Change detection was performed with Git.



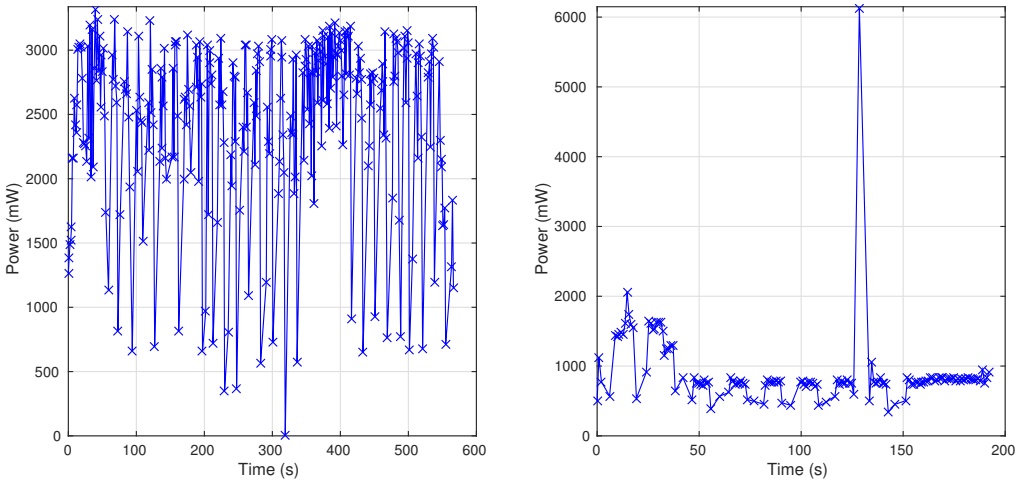
(a) CPU consumption. (b) Memory consumption.

Figure 4.14: Partial offloading (60%, n=449) memory and CPU consumption.



(a) Power usage. (b) Individual processes.

Figure 4.15: Partial offloading (60%, n=449) power usage and individual process resource consumption.



(a) 60% partial offloading power usage when suspension was enabled.

(b) Full offloading power usage when suspension was enabled.

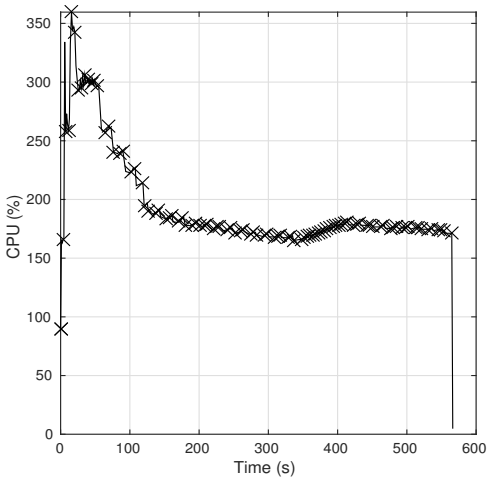
Figure 4.16: Asus Nexus 7 2013 face recognition offloading with suspension (n=449).

4.3.4 Offloading With Suspension

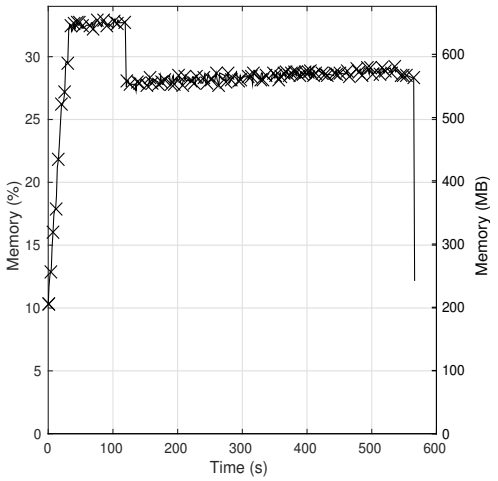
The long running time during the offloading tests caused the tablet to go into suspension mode. Executing high load tasks in suspension mode increased both latency and measurement recording interval. Measurement recording interval was observed to improve by increasing the priority of the logger utility.

Furthermore, suspension mode affected both the running time and energy consumption of the offloading tests. In Figures 4.16(a) and 4.16(b) power usage is drawn for one partial offloading tests and the full offloading test when suspension was enabled in the tablet. The tablet's idle power level was approximately 0.78W or 73% from the idle power level when suspension was not used (1.07W). In terms of running time the partial offloading test with suspension took 42% longer time and the 20, 40 and 60% tests used 0.2-1% less energy than without suspension support. With full offloading the energy savings were approximately 20%, while running time increased by 2%.

The tablet executed the task using on average less CPU than without suspension as presented in Figures 4.17(a) and 4.18(b). Finally, the tablet initially reserved on average less memory than without suspension, until the memory level reached the same levels than in experiments without power-saving mode as illustrated in Figures 4.17(b) and 4.18(b).

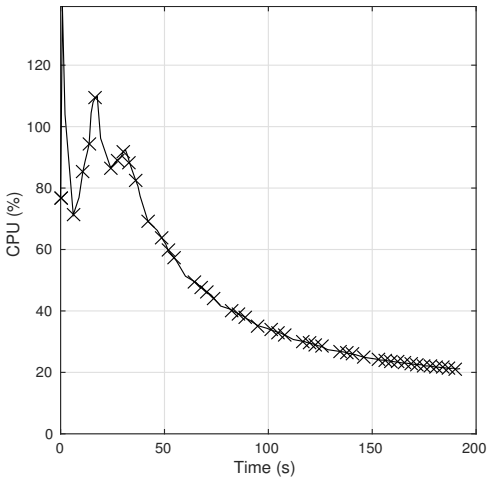


(a) 60% partial offloading CPU usage when suspension was enabled.

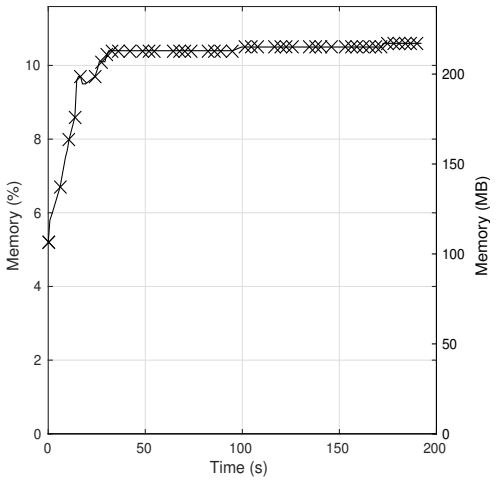


(b) 60% partial offloading memory usage when suspension was enabled.

Figure 4.17: Asus Nexus 7 2013 face recognition CPU and memory usage with suspension in 60% partial offloading test.



(a) Full offloading CPU usage when suspension was enabled.



(b) Full offloading memory usage when suspension was enabled.

Figure 4.18: Asus Nexus 7 2013 face recognition memory usage with suspension in full offloading test (n=449).

4.3.5 Varying The Number of CPUs

In order to improve energy efficiency of the parallel computation, it is possible to run the local computation by varying the amounts of cores used for the task. This reduces the performance of the local execution and thus increases its running time. This may be favoured in case the lowered performance results in that the local computation process finishes approximately at the same time as the offloaded computation while the local computation uses less energy in total than run with a higher number of cores and waiting for the offloaded computation to complete.

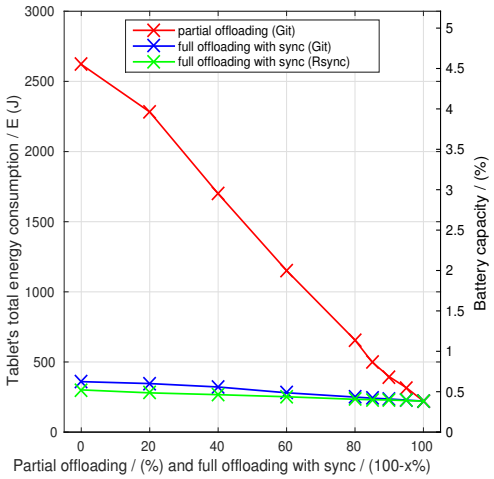
Thus, we also run the 90% partial offloading test by configuring the local worker to use two CPUs instead of three CPUs. The tablet consumed slightly higher amount of energy (4.1%) and time (3.2%). In this configuration the processor temperature remains lower than in the configuration of using three CPUs. In conclusion, using higher number of cores for high load tasks were more energy efficient than running the same task with fewer cores.

4.3.6 Offloading Summary

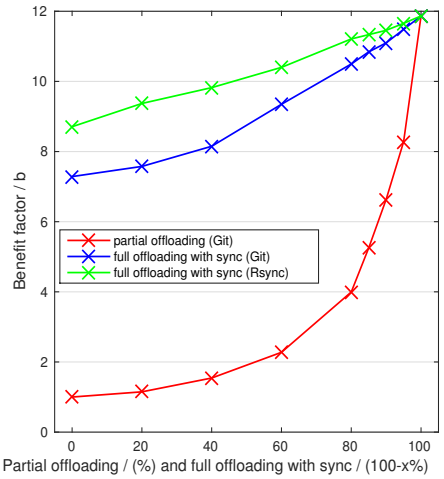
Figure 4.19(a) presents the application's energy usage for partial offloading and full offloading with data set synchronization. Figure 4.20(a) presents respective running times, which shows that for this application, the shortest running time is used with approximately 90% partial offloading. Figure 4.21(b) shows the tablet's power consumption in this configuration, while Figure 4.21(a) presents 85% offloading scenario when remote and local execution are completed around the same time.

In terms of consumed energy in face recognition, offloading with data synchronization was clearly favored compared to partial offloading and local computation with new files in each experiment as shown in Figure 4.19(b). Both experiments included possible power draw consumed during waiting for receiving results back from the server. The largest benefit factor for partial offloading was 8.3 in case of 95% of images offloaded. The benefit factors drop significantly for each time the amount of offloading is lowered (6.5 for 90%, 5.3 for 85% and 4.1 for 80%) while respective values for offloading with synchronization for both Git and Rsync are between 11.5 and 10.5. Both synchronization methods compared well against partial offloading. Git's benefit factor varied between 7.3 and 11.3 while Rsync's respective value varied between 8.6 and 11.5.

In terms of running times in Figure 4.20(b), approximately 90% partial offloading proved to have the highest performance with 5.5 benefit factor. Also both 85% and 95% partial offloading experiments had higher perfor-

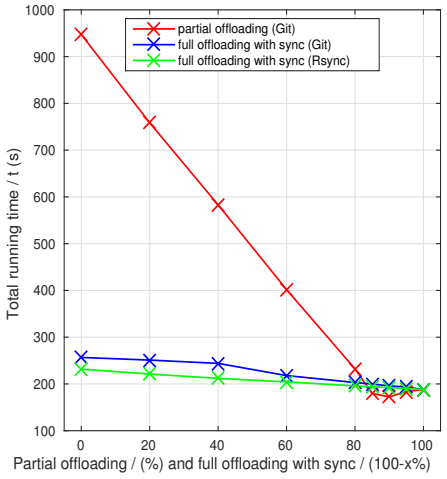


(a) Energy consumption.

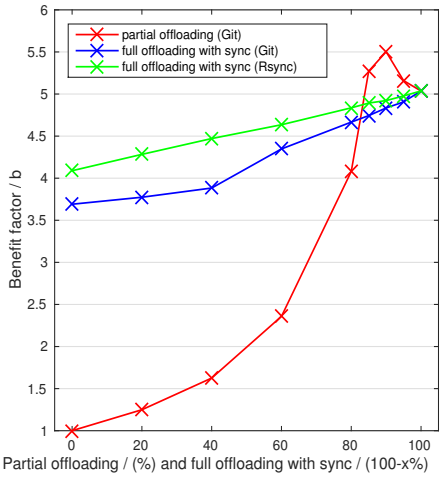


(b) Benefit factors in terms of energy consumption.

Figure 4.19: Asus Nexus 7 2013 face recognition partial offloading and full offloading with synchronization's energy consumption.

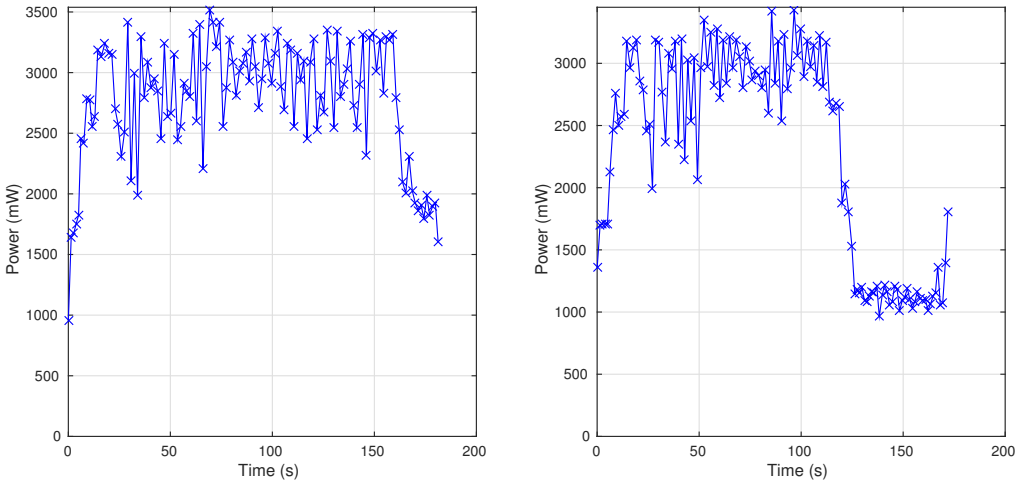


(a) Running time.



(b) Benefit factors in terms of running time.

Figure 4.20: Asus Nexus 7 2013 face recognition partial offloading and full offloading with data set synchronization's running time.



(a) 85% partial offloading power usage.

(b) 90% partial offloading power usage.

Figure 4.21: Asus Nexus 7 2013 face recognition 85% and 90% partial offloading power usage.

mance than full offloading with synchronization using either Git or Rsync with benefit factors of 5.3 and 5.2 respectively. When synchronizing new data and offloading the tasks to the server the benefit factor in terms of time varied between 4.1 and 5 with Rsync and respectively between 3.7 and 4.9 with Git.

Figure 4.22 presents the total average resource consumption for different offloading approaches. The resource utilization values were obtained as the sum of each process' resource consumption weighted by its running time with respect to the total running time of the test. With partial offloading either the local execution or the offloaded execution may complete first. The point when both local and offloaded execution were finished at approximately the same time occurred with 85% offloading when using one external worker. Thus, this test has the highest average resource consumption. With over 85% offloading the resource usage levels decreased due to the local executor completing its work earlier than the offloaded worker. On the other hand, with higher amount of local processing the cloudlet driver completes earlier than the local worker releasing resources.

4.3.7 Optimizations

As the data set used in the Caltech experiments was a collection of binary files, Git's performance can be improve by disabling compression. An experiment was run three times using a food crop image data set of consisting of 103

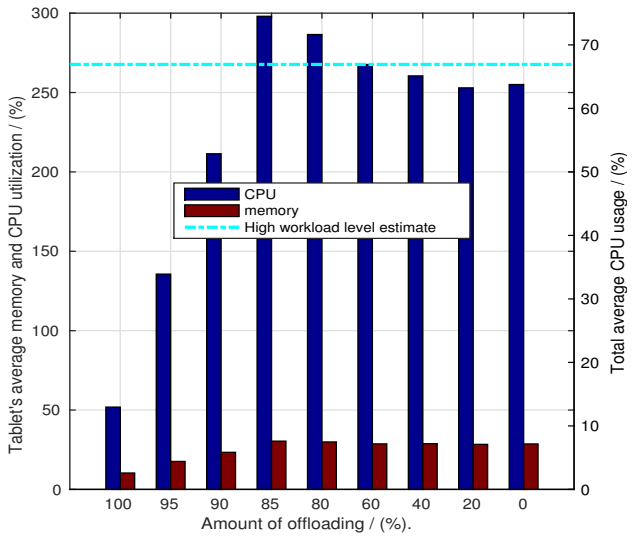


Figure 4.22: Tablet's total average CPU and memory consumption in various offloading tests.

JPEG pictures with a size of 28 MB. This experiment was conducted with Git 1.7.11.1 in Apple Mac Leopard laptop with dual core Intel core clocked at 2GHz and 1GB of memory. The results for *git add*-phase indicated that its performance was improved by over 60% in terms of running time when compression was not enabled.

For mixed repositories consisting of both binary files and text files Git uses the same compression level for all files defined in the `core.compression` attribute. Thus, turning this property off disables also compression of text content. On the positive side, if the binary contents are stored to Git repository, it can be configured to apply a suitable compression level for the deltas on given file types, which may increase performance and reduce energy costs. An alternative method for mixed repositories is to use Git for managing the versioning of the binary files' names and metadata while using a submodule, such as *git annex*[3] and *bup*[4], to store the content of the binary files to a seperate location from the shared repository.

Another optimization for performing offloading is to allow the tablet to use suspension during full offloading or after the tablet has finished its local task and needs to wait for the offloaded task to complete. As discussed in 4.3.4 this improves energy efficiency and increases slightly total running time of full offloading. For partial offloading tests, it was observed that when using

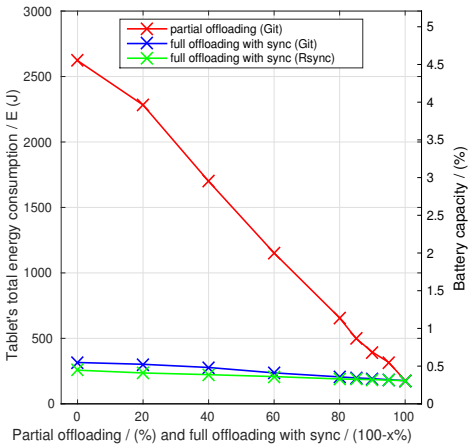
[3]http://git-annex.branchable.com/. Accessed: 11th February 2015
[4]https://github.com/apenwarr/bup. Accessed: 11th February 2015

suspension mode the energy efficiency decreased after the tablet's workload raised over the workload level of the 60% partial offloading test as shown in Figure 4.22. The energy efficiency of the 80% partial offloading experiment was 2-12% less than running the test by issuing an active request. In this case, performance was also improved without suspension state. This lead to the decrease of the time the tablet's CPUs operated at a high load level and thus, improved the experiment's energy efficiency.

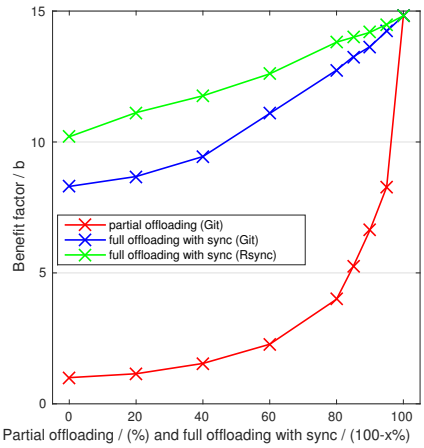
Figure 4.23(a) summarizes the energy costs for full offloading with synchronization and partial offloading, while Figure 4.23(b) shows the respective benefit factors. With this optimization, offloading with Rsync used less than ten times energy compared to execution on the tablet, while energy savings were eight times improved when using Git respectively.

The face recognizer application utilized in the full offloading test on average in total 210 MB of main memory and 12.5% of the tablet's total CPU resources (Figure 4.6(b)). The most memory intensive process was the master or standalone scheduler and the most CPU intensive process was the cloudlet driver. In case of using full offloading, instead of executing the Spark framework on the phone, a Spark job could be submitted to a Spark job server running the Spark Context to release the phone's resources [12]. Running the driver programs within the same Spark Context also allows the programs to share common libraries and thus reduce memory requirements. When the jobs process same data sets, they benefit from a framework, such as Tachyon[5], which caches RDDs. On the phone, the application can also be submitted to a Spark job server.

[5]http://tachyon-project.org/. Accessed: 4th February 2015.



(a) Energy consumption.



(b) Benefit factors in terms of energy consumption.

Figure 4.23: Asus Nexus 7 2013 face recognition partial offloading and full offloading with synchronization's energy consumption when suspension was enabled.

4.4 Linux

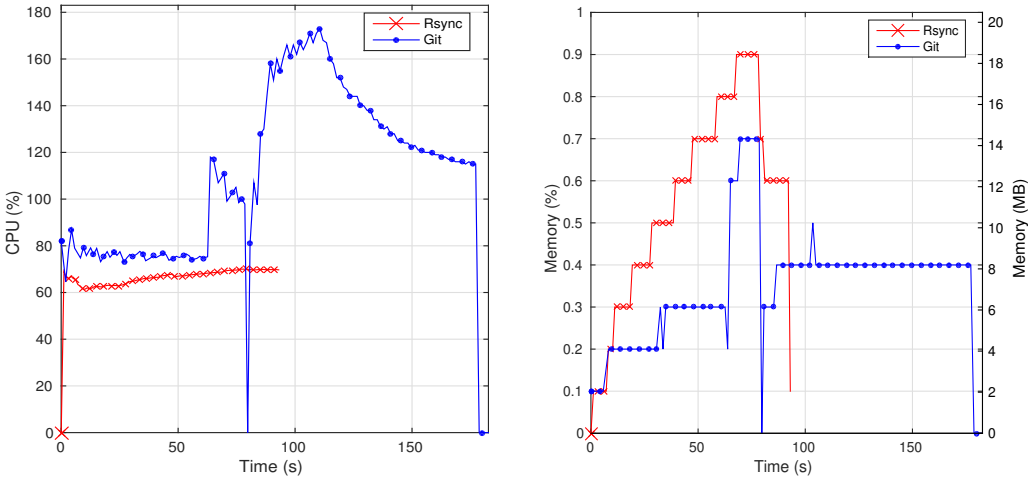
In this chapter a series of synchronization tests using Linux data sets is conducted with the Nexus 2013 tablet. The Linux versions and the corresponding data sets' uncompressed sizes are summarized in Table 4.2. The size of changes between the data sets were obtained by synchronizing the data sets without compression with Rsync. Before each test was launched the tablet was instructed to disable suspension with *powerd-cli*-utility by issuing an active request similar to the offloading and Caltech data synchronization tests. In experiments using Git the expended energy by synchronization phase was computed by integrating power over each phase's time interval.

Table 4.2: Linux data sets used in synchronization tests on Asus Nexus 7 2013.

test name	**size A (MiB)**	**size B (MiB)**	**diff (MiB)**	**# transferred files**	**# created files**	**# deleted files**
Linux full (A=2.6.32.63)	435	-	-	46, 800	46, 800	-
Linux upd 1 (A=2.6.32.63, B=3.15.1)	435	651	512	46,800,	25,100	8,900
Linux upd 2 (A=3.15.1, B=3.17)	651	657	274	14,800	2,000	1,300

4.4.1 Full Synchronization

The first Linux experiment performed a full data transfer of Linux 2.6.32.63 OS using Nexus 2013 tablet. We measured CPU and memory consumption, as well as power usage. As Figure 4.25 shows, Rsync used more power during the same period of time than Git as Rsync both compresses each file utilizing the reference file [62] and uses the tablet's network interface. During this period Git compresses the files and creates the commit, but has not yet started the file transfer phase. On the other hand, as Figure 4.24(a) presents, Rsync used less CPU compared during the same period. In the beginning of push phase, Git creates a pack file utilizing multiple threads, which explains the



(a) CPU consumption. (b) Memory consumption.

Figure 4.24: Tablet's full Linux 2.6.32.63 synchronization.

high CPU usage. Finally, Figure 4.24(b) illustrates that Rsync used higher amount of memory for storing the block fingerprint tables, compression and data transfer than Git for compression and creating the commit.

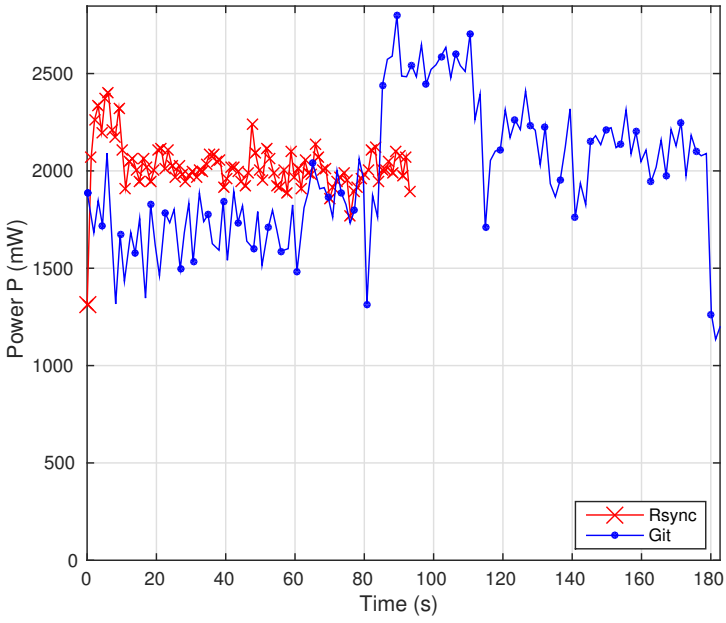
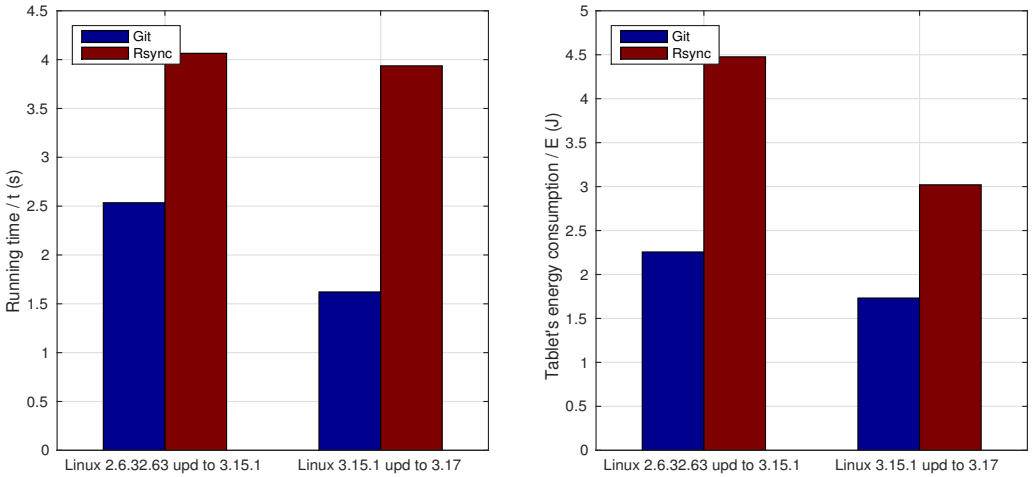


Figure 4.25: Tablet's full Linux 2.6.32.63 synchronization - Power usage.



(a) Running time. (b) Energy consumption.

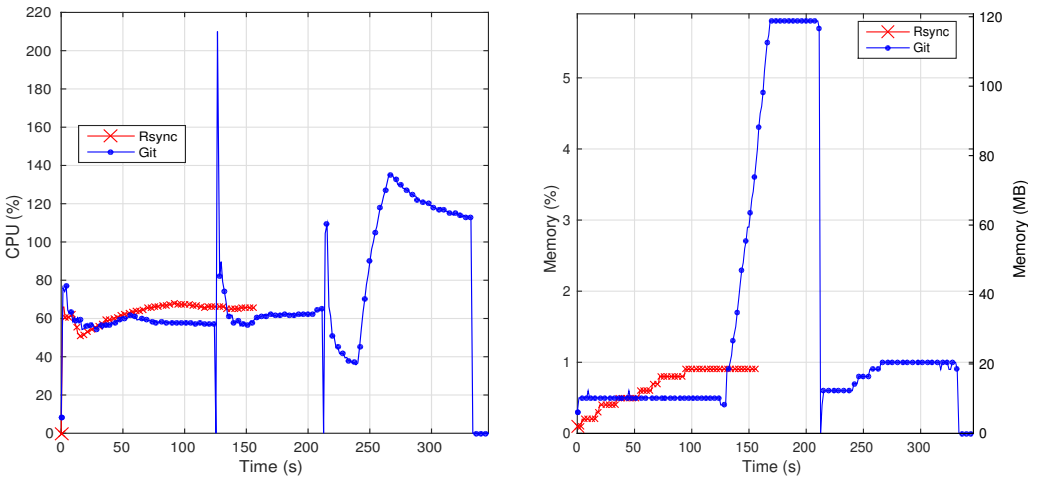
Figure 4.26: Linux changed files list building time and energy usage. Both test sets were run on Asus Nexus 7 2013 tablet.

4.4.2 Update Detection

The times for constructing the list of changed files since the previous synchronization time are summarized for both Linux update tests in Figure 4.26(a). Git compares the index to the working directory to find out modified files and the HEAD commit to the working directory to search for new and deleted files. Git solves the set of changed files significantly faster than Rsync as it has access to the change history in the tablet, while Rsync uses the network to transfer the changes between the tablet's and the remote machine's directory tree. To compare the time for starting the synchronization, Git tests would need to add the cost of one round trip between the server and the mobile device [28] and the computation of the changes between the HEAD commit and its child commit.

4.4.3 Update Synchronization

The first update test synchronized Linux 2.6.32.63 version to Linux 3.15.1 version. The results for CPU in Figure 4.27(a) and power usage in Figure 4.28 indicate that the amount of total synchronization time and energy increased compared to the full Linux transfer. In contrast to the full Linux synchronization test, memory consumption for Git was higher than for Rsync as shown in Figure 4.27(b) during the commit phase (130-210 s). Git appears to reserve memory proportionally to the size of the staged files. Git used



(a) CPU consumption. (b) Memory consumption.

Figure 4.27: Linux 2.6.32.63 version update to Linux 3.15.1.

highest amount of energy for computing the deltas and delta compression utilizing multi-threading during the start of the push phase.

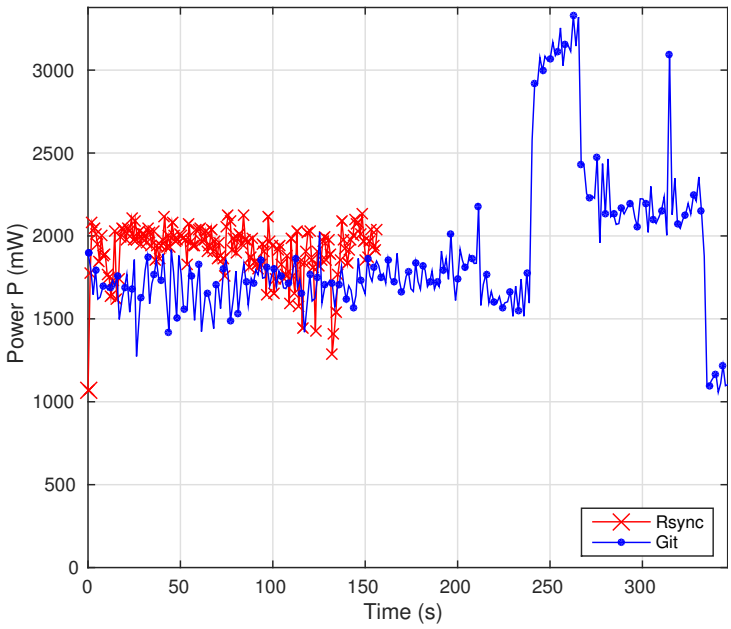
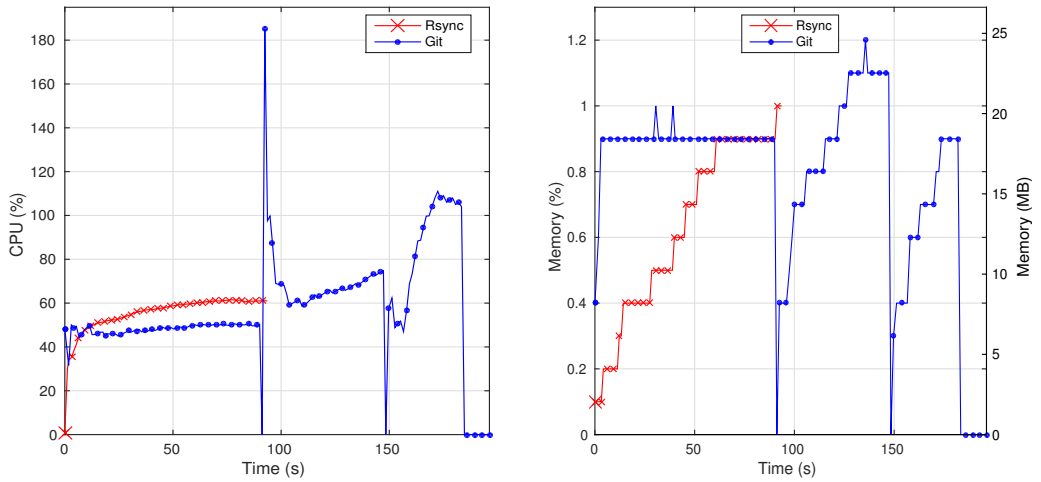


Figure 4.28: Linux 2.6.32.63 update to Linux 3.15.1 - Power usage.

In the second update test Linux 3.15.1 was synchronized to version 3.17. Measurement results for CPU, memory and power usage are shown in Fig-



(a) CPU consumption.

(b) Memory consumption.

Figure 4.29: Linux 3.15.1 update to Linux 3.17 CPU and memory usage.

ures 4.29(a), 4.29(b) and 4.30. Following the trend set by the full Linux synchronization experiment and the first update experiment, Rsync used higher amount of power and less CPU compared to Git during the same period of time when Rsync in addition to computing the deltas, compresses and transfers the changes.

The use of multiple threads for change processing improved Git's performance. Git managed to compute the changes, compress them, generate the pack file and transfer the data in less than 50 seconds while Rsync's whole process took around 90 second as shown in Figure 4.31(b). As summarized in Figure 4.31(a) Git used for change processing and transfer 2.06X less energy and 1.93X less time compared to Rsync. Git tracks files by their unique SHA hash, thus enabling it to detect moved files without needing to transfer them. Furthermore, Rsync may be configured to search missing files from a destination directory.

4.4.4 Summary

The Rsync algorithm is more computation intensive at the sender side compared to the receiver side. On the other hand, as the file is reconstructed at the receiver, the sender uses less disk I/O and main memory than the receiver. By comparing the power usage levels between the Linux tests network costs with high bit rate in the full Linux test draw slightly more power than used for computation and network transfer with lower average bit rate in the update tests.

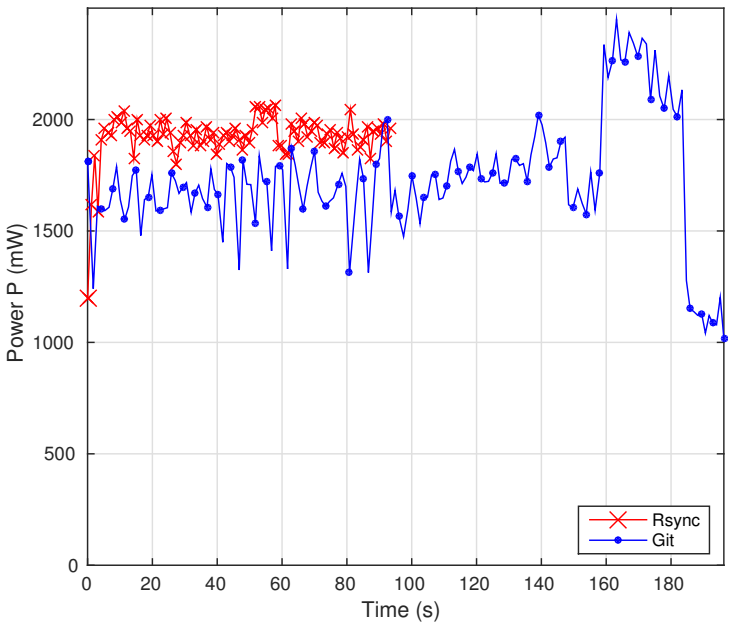
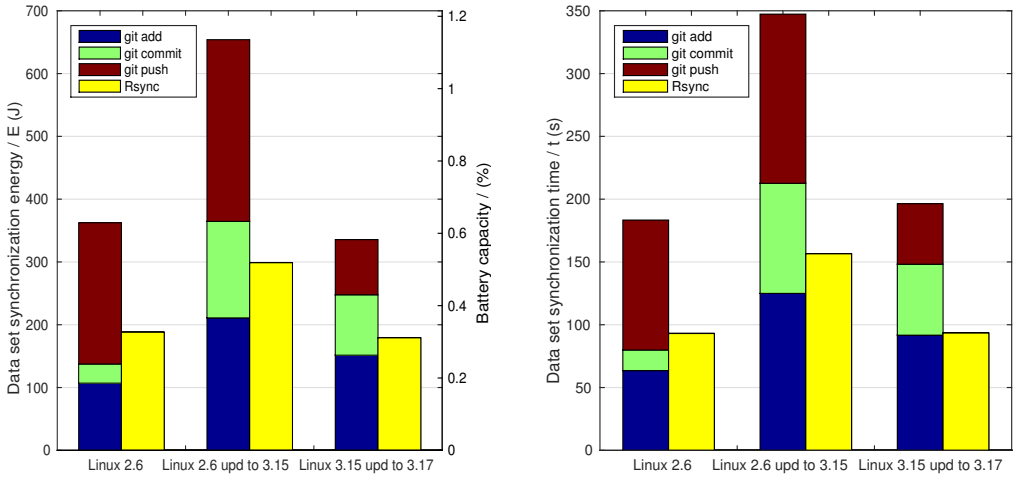


Figure 4.30: Linux 3.15.1 update to Linux 3.17 power usage.

Figure 4.31(a) presents total energy usage for Linux tests and Figure 4.31(b) shows total running time in different synchronization phases for the respective tests. The first Linux update test used higher amount of time and energy compared to the full synchronization test for both Git and Rsync. Git spends a significant proportion of time in adding the changed files to the repository, as it writes a snapshot of each changed file. Further, the push phase takes longer time in the first update test compared to the push phase in the full synchronization test due to computation of the deltas between the two commits and the compression of the deltas.

The first update test indicated that computing the changes can use slightly higher amount of energy and time compared to performing a full data set synchronization. in Figure 4.31(a) Rsync used nearly 1.58X energy in the first update test compared to the full Linux test while the size difference between the distributions is 1.5X according to Table 4.2. The total size of changes is as high as 78% of the first update test's data set. Possible reductions in energy usage or time usage would be offset by increase in network traffic and subsequently affecting other network users and increasing power usage of the network path's components.

Both Rsync and Git used slightly less energy in the second update test compared to the full Linux test while the amount of transferred data was 63% (275 MiB) of the size of the full Linux distribution. As all energy expenditure



(a) Energy consumption.

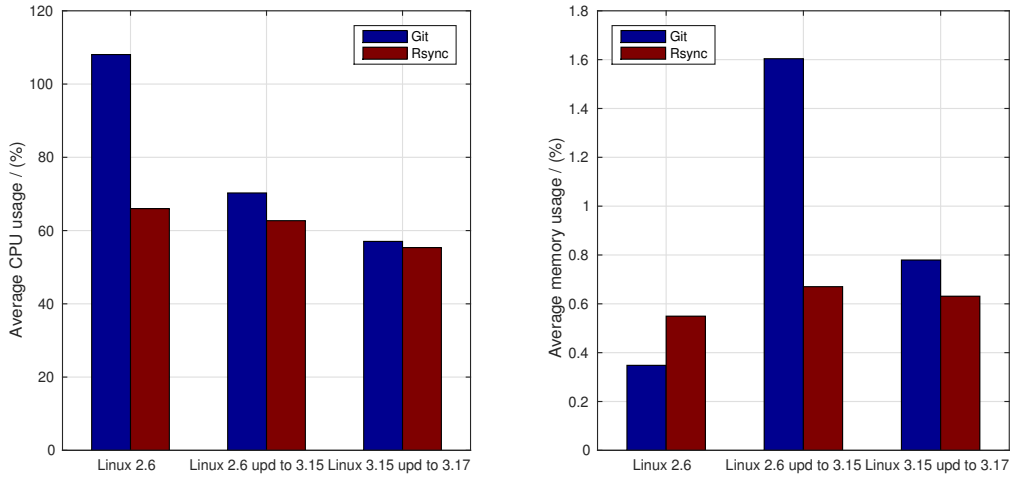
(b) Running time.

Figure 4.31: Linux synchronization total energy and running time usage.

in the full Linux tests is consumed in compression and network file transfer, computing the changes in the second Linux update tests took around 37% of total energy consumed. Around 15,000 regular files were transferred in the second update tests, while the full Linux test consisted of around 47,000 regular files. Assuming linear increase in energy usage with respect to the size of the transferred data, the benefit of sending only the changes in the second update tests was around 1.51 for both Git and Rsync. Thus, the network cost dominated over computation costs when the size of the data sets changes was not over one third of the total size of the sender's data set.

The overall energy usage from the tablet's battery capacity for Git was around 0.63% in the full Linux test, 1.14% for the first update test and 0.59% for the second update test, while Rsync's respective energy usage levels were 0.33%, 0.52% and 0.31%. Git also performs operations necessary for distributed version control which necessitates the use of higher amount of resources compared to Rsync.

In the Linux update synchronization tests Git used higher amount of CPU and main memory compared to Rsync as is shown in Figure 4.32(a) for CPU usage and Figure 4.32(b) for main memory usage. Rsync had higher main memory usage rate than Git in the full synchronization test. Git utilizes multi-threading for change processing and packs the changes into single file, which makes Git more resource intensive compared to using one Rsync process.



(a) CPU consumption. (b) Memory consumption.

Figure 4.32: Linux synchronization total average CPU and memory usage.

Chapter 5

Discussion

5.1 Energy efficiency of data synchronization

In this Thesis, we have studied several factors that affect the cost of data synchronization. The first was the latency of communication channel. The sender is required to wait acknowledgment from the receiver every time the receiver's receive window is consumed. This wait time increases with high latency. This can be alleviated by using PSM, though as the latency improves energy consumption of the radio interface may become higher in this mode [33]. Another aspect affecting the energy efficiency of communication was studied in [53], where the authors found that higher transfer speed resulted consistently to higher energy efficiency of transferring data for both WLAN and 3G connections. The study also illustrates that for WLAN connections using the same bit rate a bursty traffic pattern lead to energy efficiency improvements over a smooth traffic pattern.

The energy efficiency wireless connection depends on the use context. For example, the study in [43] revealed that 3G consumed less in terms of battery usage compared to connecting the mobile device to a public WLAN hot spot. To summarize, the decision to perform data synchronization is favorable during low congestion time intervals with strong signal strength and low latency. As the experiments conducted in this Thesis were done in low latency, high speed wireless network, the context had a favorable effect on the measured running time and energy consumption.

The order of two-way synchronization is essential for efficient synchronization and communication. The mobile device should first synchronize its data with the remote source before sending its own changes to the remote source as changes written to same files by both parties are typically rejected by the remote client. Thus, the first stage of syncing consists of merging

the changes at the mobile client and the second stage of sending the local changes to the remote party. The first stage of syncing is often called a pull operation and the second stage a push operation. This approach allows to resolve conflicts on the parts of data that have diverged.

With Ubuntu Touch running in the Nexus 7 2013 tablet, use of suspension mode resulted in higher running times and lower energy efficiency for high complexity tasks. This was significant for example in the Caltech data set synchronization tests, in which one experiment used one CPU, while the other tests used three CPUs. Conducting the 90% partial offloading test with two cores also resulted in reduced energy efficiency than with three cores. We estimated that the tablet's energy efficiency was improved when preventing suspension mode for tasks, which required over 67% of total CPU computing resources.

We measured the performance cost for building a list of changed files during the start of the synchronization in the Linux update experiments. This may be a significant issue for whole file system synchronizations for instance in the case of performing an incremental backup of the mobile device. One solution is to log file change events with *inotify*-utility which is implemented for example in CDroid [24]. Implementing directory tree change logging may prove difficult due to the requirement that the synchronization software needs permissions for reading the phone's applications data folders. The authors in [25] rooted the phones for gaining the access to place file directory watches for Android phones applications. Another approach for performing large one-way directory synchronization is to use file systems supporting incremental backups, such as Btrfs [6] and ZFS [16], though to use Btrfs the mobile device must support Linux kernel 3.17 or later.

The synchronization tool's protocol also affects the energy efficiency of the data transfer. Both Rsync and Syxaw utilize pipelining which is defined in this context as using one logical round-trip for multiple files [62]. The benefit of pipelining is emphasized in high latency communication environments. The used protocol also has an effect on the size of the header content. The higher the number of files is, the greater the effect of compact header content. Finally, the protocol should aim to minimize the required number of two-way communication rounds due to cost of latency. Rsync for example, solves this by choosing the file's block size depending on the file's size and sending the file's block signatures in one round. Another alternative is to employ multi-round synchronization, in which the algorithm starts by first computing high level fingerprints that represent super chunks of the file and recursively dividing these high level chunks into smaller chunks on each successive communication round [41]. Multi-round synchronization can reduce bandwidth usage by a factor of 2 or 3 when the hosts avoid sending some of

the fingerprints which mask the additional increase in latency and protocol overhead [41].

The cost of data transfer and storage was discussed in [30]. The authors of the study emphasize the different costs for data transfer depending on target users needs. With higher reliability and performance requirements, the network costs may become only feasible for medium and large organizations. The authors also go through an example computation of evaluating the cost of secure transmission using different techniques and secure data storage.

According to research by Barbera et al. [25] synchronization frequency also affects both the amount of generated traffic and energy cost of the mobile device. The authors studied the costs of synchronizing application data and user created data for 11 participants during three weeks experiment. They conclude that for less frequent synchronization periods both the amount of traffic and energy usage decreased. For applications generated data the difference in the amount of transferred data was especially high due to a high number of temporary files. The amount of user created data was around four times lower compared to application generated data while the energy costs for synchronizing the two data types were much less pronounced. According to the authors costs of change computing dominated over file transfer costs, and thus the difference in energy costs between the two data types were attenuated. Total battery usage per day for the application and user generated data using 5 min interval was approximately 36% and 53%. For the 30 min (2h) interval synchronization cost for the application data was 11% (2.7%) of the battery and for the user created data around 8% (2.3%) of the battery.

Git and Rsync take a different approach with respect to transferring the changes. While Git transfers the data after computing the changes and compressing the changes in the end of push phase, Rsync uses the network in a intermittent manner sending each time the current file's accumulated changes when the sender finds a common block between its version file and the receiver's basis file. According to the research in [53], higher bit rates lead to higher energy efficiency of the data transfer, which implies that running several Rsync instances in parallel may result to improved usage of network connection and reduced energy consumption.

5.2 Business opportunities

In [58] the authors discuss the use of cloud services from the viewpoint of trust and security. This is especially remarkable for mobile browsing offloading, where the cloud server or SPDY gateway may be able to track down users page loads. In [24] the authors implemented the secure communica-

tion component for HTTPS connections separately from HTTP communication module. Thus, they were able to give the user more options regarding safe web browsing, such as tunneling certain HTTP connections through the server to disable header checking for compression. The authors of CDroid continue that their cloud server does compromise user's security in order to compress and detect user information leakage. In case of using secure sockets layer (SSL) it does this by acting as a certificate authority and passing its own certificate to the phone, which already includes these certificates. Thus, the cloud server can then decrypt users' data. The authors note that this technique is applicable for most mobile applications, except certain applications that verify the certificate authority against their own certificate storage. The users may consider the benefits gained from reducing the amount of traffic and battery usage, as well as the benefits and possible disadvantages in terms of user privacy in using cloud providers services or SPDY gateways to leverage mobile browsing offloading.

In addition to securing the connections using either SSH or SSL, the service provider needs to provide means to identify their customers. An example authentication implementation is overviewed in [58], in which the authors use simple authentication and security layer (SASL) to support several authentication mechanisms. By identifying the user the cloud can for example provide higher amount of disk space by managing quotas for different users and tracking information for billing and statistics. For cloudlet architectures users' data has to be synchronized periodically with the central cloud.

Satyanarayanan et al. discuss two possible deployment models for mobile offloading service providers [58]. In the first category deployment is driven top-down and the cloud infrastructure owner and service providers do not share the same business goals as the service providers rent resources from the cloud owners. In contrast, the second deployment model is driven bottom-up as the cloud infrastructure owners both plan and setup the cloud system and services.

While the implementation of any mobile offloading service ideally supports a wide range of wireless technologies, the deployment of these systems are constrained based on environment specific regulations. For example, the Bioinitiative report [27] published as a result of joint effort of international researchers in their specific field, summarizes and in some cases presents unpublished research results of the effects of radiofrequency (RF) radiation (mostly of wireless technologies used in cellular phones and WiFi) and extremely low frequency (ELF) radiation produced in the transfer of electricity and use of electricity appliances on biological organisms (mainly human and animal studies). It focuses on biological and functional effects of low intensity ELF and RF electromagnetic radiation at the cellular level and at the organ-

ism's physiological level. Several mechanisms are illustrated. For example, Martin Blank describes how by changing the frequency of the cell's action potential (event of energy exchange) different proteins can be synthesized. ELF and RF radiation have also been successfully used in therapeutic applications, for example in treating bone reunions with pulsed magnetic fields since 1979 in USA. Theoretical background of bio-sciences and cell regulation are given in [35, 59] whose author, Georgi Stankov, discovered the Universal law of nature in relation to cell metabolism.

Authors of the Bioinitiative report reason that public safety limits for ELF and RF electromagnetic radiation set by International Commission on Non-Ionizing Radiation Protection (ICNIRP) in 1998 and Federal communications commission (FCC) in US should be updated to correspond the studies conducted by the international research community since 1998 and the standard should be based on biological effects instead of the levels required for causing thermal effects [27]. The report defines and gives recommendations for using the principle of preventive precaution in deployment and use of wireless, as well as wired systems and appliances. It also defines recommendation levels based on biological effects for ELF electromagnetic radiation and RF radiation that take into account in addition to exposure intensity and duration, for example pulse modulation of the carrier wave and different demographic groups. Several teacher federations and unions for example in USA, Canada, United Kingdom and Germany already recommend wired technologies in public schools [18]. However, alternative wireless technologies are pointed out for example in Council of Europe's report 2011 which utilize optoelectronics that use frequencies in the visible light and infrared range [40].

The synchronization tools were tested in the Ubuntu Touch OS and may need further experimenting with the Android OS. There are several Git clients implemented for Android that support both read and write operations to the repository. For Rsync, Google Play store provides several Rsync clients. One alternative is to implement a custom synchronization software using the Rsync algorithm. This was done for example in [25], whose authors implemented Rsync's rolling hash function for their synchronization module.

Chapter 6

Conclusions

In this Thesis, mobile offloading and data synchronization was studied in terms of running time and used energy. Partial offloading results showed that in terms of making the offloading decision, the user should be able to pass a target time limit for the offloading decision component for making the decision to use partial offloading in favor of performance over total energy usage.

When the mobile device is long periods of time inactive while running a background task, the device may utilize suspension mode for reducing energy usage. In the full offloading test this resulted in energy savings with slight additional cost to the task's running time. On the other hand, in the partial offloading tests with suspension performance was observed to be significantly reduced, while in most of these tests energy usage levels decreased slightly. For high load tasks conducted in both data synchronization tests and partial offloading experiments, energy efficiency was observed to improve when preventing the system to go into suspension. Thus, the user or user application should be able to have the option to change suspension mode.

Furthermore, memory, CPU and power usage profiling was performed for both the Caltech and Linux synchronization experiments. Rsync had a lower memory footprint and CPU consumption than Git in most of the experiments. This results in more energy efficient synchronization in mobile environment which stores the master copy of the data source. Git has full support for two-way synchronization and it maintains a version history of all changes made to each file in the version control repository. This leads to the use of higher amount of CPU and memory during synchronization and increase in the amount of secondary storage space. On the other hand, Git supports the move operation, which is not supported by default in Rsync. In case of one-way synchronization, Git also performs file change detection much faster than Rsync as it can utilize the local repository.

The cost of flexibility to enable two-way synchronization involves storing a snapshot of the changed files for each time a change set is checked into the repository. For instance, Git creates a new snapshot of each changed file that is added to its repository. For performing efficient synchronization, Git computes the deltas for each file between successive commits since the last synchronization round. Changes are stored into compressed format. Mercurial [10] and Unison [55] are other synchronization tool examples that manage snapshots. Unison also allows to choose the files which the user wants to be able to merge changes from the remote source.

Chapter 7

Bibliography

[1] Apache Spark https://spark.apache.org/. Accessed: 14th of September 2014.

[2] Akka Framework. http://akka.io/.

[3] Apache Hadoop. http://hadoop.apache.org/.

[4] The archive of the Computational Vision Group at Caltech. http://www.vision.caltech.edu/html-files/archive.html. Accessed: 8th of December 2014.

[5] Asus Google Nexus 7 2013 specifications. http://www.gsmarena.com/asus_google_nexus_7_%282013%29-5600.php. Accessed: 19th January 2015.

[6] Btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/Main_Page. Accessed: 7th of December 2014.

[7] inotify manual pages. http://man7.org/linux/man-pages/man7/inotify.7.html. Accessed: 8th of January 2015.

[8] Mac basics: Time machine backs up your Mac. http://support.apple.com/en-us/HT201250. Accessed: 16th of December.

[9] Manual page for redir utility. http://manpages.ubuntu.com/manpages/precise/man1/redir.1.html. Accessed: 7th of December 2014.

[10] Mercurial. http://mercurial.selenic.com/. Accessed: 9th of January 2015.

[11] Nokia Xpress. http://www.windowsphone.com/en-us/store/app/nokia-xpress/cbf5f827-aa0a-4670-8ba6-775676f275b0. Accessed: 22nd of October 2014.

[12] Ooyla - Spark as a service. https://github.com/ooyala/spark-jobserver. Accessed: 24th of October 2014.

[13] OpenCV. http://opencv.org. Accessed: 5th of December 2014.

[14] Opera Mini. http://www.opera.com/mobile/. Accessed: 22nd of October 2014.

[15] Scala. http://scala-lang.org/. Accessed: 14th of September 2014.

[16] Sending and Receiving ZFS data. http://docs.oracle.com/cd/E19253-01/819-5461/gbchx/index.html. Accessed: 14th of January 2015.

[17] Single cell Li-Ion battery fuel gauge for battery pack integration. www.ti.com/cn/lit/gpn/bq27541-v200. Accessed: 7th of December 2014.

[18] Teachers' unions in Germany, UK, USA and Canada who have done their research do not support WiFi in schools. http://www.safeinschool.org/2013/03/why-teachers-unions-dont-support-wifi.html. Accessed: 12th May 2105.

[19] Ubuntu Touch. http://www.ubuntu.com/phone. Accessed: 19th January 2015.

[20] Ubuntu Wiki Powerd. https://wiki.ubuntu.com/powerd. Accessed: 7th of December 2014.

[21] What is Amazon Silk. http://docs.aws.amazon.com/silk/latest/developerguide/introduction.html. Accessed: 22nd of October 2014.

[22] Zlib home page. http://zlib.net. Accessed: 18th of September 2014.

[23] Bahtovski, A., and Gusev, M. Cloudlet challenges. *Procedia Engineering 69*, 0 (2014), 704 – 711. 24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013.

[24] Barbera, M. V., Kosta, S., Mei, A., Perta, V. C., and Stefa, J. Mobile offloading in the wild: Findings and lessons learned through a real-life experiment with a new cloud-aware system. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014* (2014), pp. 2355–2363.

[25] Barbera, M. V., Kosta, S., Mei, A., and Stefa, J. To offload or not to offload? The bandwidth and energy costs of mobile cloud computing. In *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013* (2013), pp. 1285–1293.

[26] Ben Escoto, Dean Gaudet, A. F. rdiff-backup. `http://www.nongnu.org/rdiff-backup/index.html`. Accessed: 20th of September 2014.

[27] BioInitiative Working Group, C. S., and David O. Carpenter, E. Bioinitiative report: A rationale for a biologically-based public exposure standard for electromagnetic radiation. Available at `www.bioinitiative.org`, December 2012.

[28] Chacon, S. *Pro Git*. August 2009. `http://git-scm.com/book`. Accessed: 17th of September 2014.

[29] Chen, G., Kang, B.-T., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., and Chandramouli, R. Studying energy trade offs in offloading computation/compilation in java-enabled mobile devices. *IEEE Trans. Parallel Distrib. Syst. 15*, 9 (Sept. 2004), 795–809.

[30] Chen, Y., and Sion, R. Costs and security in clouds. In *Secure Cloud Computing*. 2014, pp. 31–56.

[31] Chun, B., and Maniatis, P. Augmented smartphone applications through clone cloud execution. In *Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems, May 18-20, 2009, Monte Verità, Switzerland* (2009).

[32] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 301–314.

[33] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 49–62.

[34] Ding, Y., Savolainen, T., Korhonen, J., Tarkoma, S., Hui, P., and Kojo, M. Nao: A framework to enable efficient mobile offloading.

In *Proceedings of the Workshop on Posters and Demos Track* (New York, NY, USA, 2011), PDT '11, ACM, pp. 8:1–8:2.

[35] Georgi, S. *The General Theory of Biological Regulation The Universal Law in Bio-Science and Medicine, Volume III.* New European Academy Press, München, Sofia, 1999.

[36] Gordon, M. S., Jamshidi, D. A., Mahlke, S., Mao, Z. M., and Chen, X. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 93–106.

[37] Hallis, F., Holmbacka, S., Lund, W., Slotte, R., Lafond, S., and Lilius, J. Thermal influence on the energy efficiency of workload consolidation in many-core architecture. In *Proceedings of the 24th Tyrrhenian International Workshop on Digital Communications* (2013), R. Bolla, F. Davoli, P. Tran-Gia, and T. T. Anh, Eds., IEEE, pp. 1–6.

[38] Han, B., Hui, P., Kumar, V. S. A., Marathe, M. V., Shao, J., and Srinivasan, A. Mobile data offloading through opportunistic communications and social participation. *IEEE Transactions on Mobile Computing 11*, 5 (May 2012), 821–834.

[39] Huerta-Canepa, G., and Lee, D. A virtual cloud computing provider for mobile devices. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond* (New York, NY, USA, 2010), MCS '10, ACM, pp. 6:1–6:5.

[40] Huss, M. J. The potential dangers of electromagnetic fields and their effect on the environment. Tech. rep., Luxembourg, May 2011.

[41] Karppanen, J. Lossless differential compression for synchronizing arbitrary single dimensional strings. Master's thesis, University of Helsinki, Faculty of Science, Department of Computer Science, October 2012.

[42] Kemppainen, M. Mobile computation offloading: A context-driven approach, Aalto University School of Science, Seminar on internetworking, Spring 2011.

[43] Kosta, S., Aucinas, A., Hui, P., Mortier, R., and Zhang, X. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012* (2012), pp. 945–953.

[44] Kumar, K., Liu, J., Lu, Y., and Bhargava, B. A survey of computation offloading for mobile systems. *MONET 18*, 1 (2013), 129–140.

[45] Kumar, K., and Lu, Y. Cloud computing for mobile users: Can offloading computation save energy? *IEEE Computer 43*, 4 (2010), 51–56.

[46] Lagerspetz, E. Dessy: Desktop search and synchronization. Master's thesis, University of Helsinki, Department of Computer Science, September 2009.

[47] Lagerspetz, E., and Tarkoma, S. Mobile search and the cloud: The benefits of offloading. In *Ninth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2011, 21-25 March 2011, Seattle, WA, USA, Workshop Proceedings* (2011), pp. 117–122.

[48] Lindholm, T. *XML-Aware Data Synchronization for Mobile Devices.* PhD thesis, Helsinki University of Technology, Faculty of Science, Department of Computer Science and Engineering, December 2009.

[49] Lindholm, T., Kangasharju, J., and Tarkoma, S. Fast and simple XML tree differencing by sequence alignment. In *Proceedings of the 2006 ACM Symposium on Document Engineering* (New York, NY, USA, 2006), DocEng '06, ACM, pp. 75–84.

[50] Lindholm, T., Kangasharju, J., and Tarkoma, S. Syxaw: Data synchronization middleware for the mobile web. *MONET 14*, 5 (2009), 661–676.

[51] Losoi, M. Low-latency computing in a heterogeneous computing environment. Master's thesis, Aalto University School of Science, 2014.

[52] Merlin, M. Doing fast incremental backups with Btrfs send and receive. http://marc.merlins.org/perso/btrfs/post_2014-03-22_Btrfs-Tips_-Doing-Fast-Incremental-Backups-With-Btrfs-Send-and-Receive.html. Accessed: 16th of December 2014.

[53] Miettinen, A. P., and Nurminen, J. K. Energy efficiency of mobile clients in cloud computing. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 4–4.

[54] Oliveira, D. A. G. d. Energy consumption and performance of HPC architecture for Exascale. Master's thesis, UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2013.

[55] Pierce, B. C., and Vouillon, J. What's in Unison? A formal specification and reference implementation of a file synchronizer. Tech. Rep. MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[56] Potter, S. *The Architecture of Open Source Applications Volume II: Structure, Scale and A Few More Fearless Hacks.* `http://aosabook.org/en/git.html`. Accessed: 17th of September 2014.

[57] Roelof Kemp, Nicholas Palmer, T. K., and Bal, H. Cuckoo: Computation offloading framework for smartphones.

[58] Satyanarayanan, M., Bahl, P., Cáceres, R., and Davies, N. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing 8*, 4 (2009), 14–23.

[59] Stankov, G. *The Universal Law The General Theory of Physics and Cosmology, Volume II -concise version.* New European Academy Press, München, Sofia, 1999.

[60] Thomas, B., Jurdak, R., and Atkinson, I. SPDYing up the web. *Commun. ACM 55*, 12 (Dec. 2012), 64–73.

[61] Tridgell, A. News for rsync 3.0.0. `http://rsync.samba.org/ftp/rsync/src/rsync-3.0.0-NEWS`. Accessed: 20th of September 2014.

[62] Tridgell, A. *Efficient algorithms for sorting and synchronization.* PhD thesis, Australian National University, February 1999.

[63] Viola, P., and Jones, M. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (2001)*, vol. 1, pp. 511–518.

[64] Wang, X. S., Shen, H., and Wetherall, D. Accelerating the mobile web with selective offloading. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing* (New York, NY, USA, 2013), MCC '13, ACM, pp. 45–50.

[65] Wigley, J. Git from the bottom up. `ftp.newartisans.com/pub/git.from.bottom.up.pdf`. Accessed: 17th of September 2014.

[66] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[67] Zhang, X., Kunjithapatham, A., Jeong, S., and Gibbs, S. Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. *Mob. Netw. Appl. 16*, 3 (June 2011), 270–284.